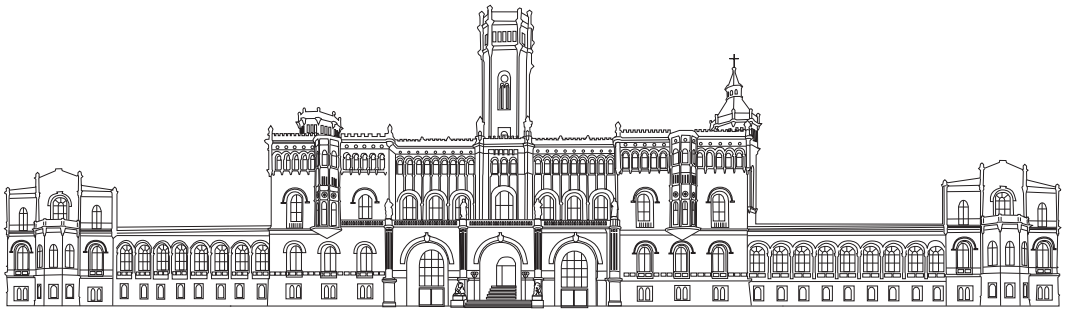
Machine Learning in Quantum Mechanical and Optical Systems



Von der Fakultät für Mathematik und Physik
der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des Grades
Doktorin der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation von

M.Sc. Viktoria-Sophie Schmiesing

2025

Referent: Prof. Dr. Tobias J. Osborne

Korreferentin: Jun.-Prof. Dr. Ramona Wolf

Korreferent: Prof. Dr. Bodo Rosenhahn

Tag der Promotion: 16.01.2025

Abstract

In recent years, machine learning (ML) has become increasingly important across science, industry, and daily life. Simultaneously, quantum computing has emerged as a field with the potential to revolutionize computation. This thesis explores the application of ML to quantum and optical systems. We present two main results: the proposal of a quantum recurrent neural network (QRNN) architecture and the use of reinforcement learning (RL) for experimental fiber coupling, a common task in quantum labs.
When data is quantum in nature, quantum ML techniques may be better suited than classical approaches. One such technique is the feed-forward dissipative quantum neural network (DQNN), which learns general quantum channels from independent and identically distributed data. However, many quantum tasks involve sequential data, such as learning quantum state evolution under time-dependent Hamiltonians or interacting with quantum environments. In classical ML, such tasks can, e.g., be handled by recurrent neural networks (RNNs). This thesis proposes a fully quantum RNN structure designed for qudits, named dissipative quantum recurrent neural network (DQRNN). Extending the DQNN framework to the recurrent case, DQRNNs can approximate general causal quantum automata. We present both quantum and classical training algorithms for DQRNNs, showing that the resource requirements scale with the width of the underlying DQNN but not with its depth. Numerical results show that DQRNNs solve memory-dependent tasks beyond the capacity of DQNNs, generalizing well from limited data. One promising future application of DQRNNs is model-based RL in quantum environments.
Although RL is inherently well-suited for control tasks, RL agents for applications in optical experiments have mostly been trained in simulation. Taking the example of fiber coupling, we show that it is feasible to apply RL directly in experiments. This saves us the time of extensive system and noise modeling. Still, intermediate challenges needed to be overcome such as time-consuming training, noisy actions, and partial observability. For shorter training times, we use a simple virtual testbed for environment tuning and algorithm selection. We demonstrate that an RL agent can learn to overcome noisy actions and partial observability. In four days of training time directly in the experimental setup, using sample-efficient algorithms such as truncated quantile critics (TQC) and soft actor-critic (SAC), it learns to achieve coupling efficiencies comparable to human experts.
This thesis takes key steps toward integrating machine learning in quantum control, introducing DQRNNs that could serve as a powerful tool for modeling quantum environments and highlighting the role of RL in experimental physics. By showing how RL can be applied successfully directly in an optical experiment using the example of fiber coupling, this work paves the way for applying RL to more intricate quantum systems.

Keywords: Machine learning, quantum computing, recurrent neural networks, reinforcement learning, fiber coupling

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Tobias J. Osborne, my supervisor, for his unwavering support and insightful guidance throughout this journey. Thank you for introducing me to the fascinating field of machine learning. I am immensely thankful for the numerous fruitful discussions, for your mentorship, and for the safe and open-minded workspace you created.

I would also like to thank Ramona Wolf and Bodo Rosenhahn for serving as referees and providing valuable feedback on my work. Special thanks to Michéle Heurs for being the committee chair and for facilitating an intriguing experimental aspect of this project that has broadened my perspective.

I gratefully acknowledge the financial support provided through SFB 1227 (DQ-mat) by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), which made this research possible.

To my research group and project partners: thank you for creating a stimulating and fun working environment. In particular, Lea Richtmann and Jan Heine, your friendship has been a wonderful bonus alongside our engaging discussions. The journey has been much more fun because of you. A special shout-out goes to the other IT admins, who were always ready to help me navigate technical challenges. Kerstin Beer, thank you for not only answering my endless questions but also for introducing me to the ins and outs of the system. Additionally, I wish to thank the Bachelor's and Master's students I have had the privilege to work with, including Robin Syring, Zi Chua, Nils Renziehausen, Nils Zolitschka, and Debora Ramaciotti. Collaborating with you has been a rewarding experience, and I have learned much from you.

I also want to thank my proofreaders: Sascha Gehrmann, Marieke Schmiesing, Lea Richtmann, Andreea Lefterovici, Jan Heine, Lennart Binkowski, Marvin Schwiering, and Kerstin Beer. Your attention to detail helped me catch countless mistakes – though, of course, I remain responsible for any that may still linger.

To Mara Meyer zum Alten Borgloh, Jan Heine, Jessica Pockrandt, and Sascha Gehrmann, thanks for the many crafting, sports, cooking, and board game sessions. Your support and friendship have meant the world to me, and I am grateful for your constant encouragement. A heartfelt thank you to my fistball team, whose camaraderie and athletic distractions offered much-needed balance and stress relief during the more intense phases of this work.

My deepest gratitude also goes to my parents, Ulrike and Johannes, and my siblings, Anne, Marieke, and Benjamin, for their unwavering love and for engaging me in fascinating discussions that have shaped my thinking in ways I cannot fully express here.

Finally, I want to thank my husband, Tobias Uhlig. Your love and support have been my foundation throughout this process. You have kept my back free, allowing me to focus on this work, and for that, I am endlessly grateful.

Thank you all.

Contents

1

Introduction

In the last years, machine learning (ML) has become more visible in our everyday life through tools [1–5] like ChatGPT [6,7], or DeepL [8]. However, the use of ML, a subfield of artificial intelligence (AI) that relies on data rather than explicit instructions to perform tasks, is not always immediately apparent in our daily lives. Applications include personalized recommendation systems [9–11], traffic estimation [12], social media algorithms [13–15], credit fraud detection [16,17], and credit scoring [18,19]. This increasing presence in our everyday life ultimately even led to the development of new laws that regulate the use of AI based on risk factors [20].

Furthermore, ML is applied in various industries and scientific disciplines, including political and social sciences [21–23], medicine [24–26] and biology [27, 28], chemistry [29–31], and physics [32–34]. In 2024, both the physics [35] Nobel prize and half of the chemistry [36] one went to ML methods and applications. As anticipated, given its broad range of applications, the field of ML includes a diverse array of methods. Generally speaking, one can divide ML into three fields: supervised, unsupervised, and reinforcement learning [37]. In *supervised learning (SL)*, specific function approximators are fit to a given set of input-output pairs [38]. This can, for example, be applied to breast cancer diagnosis [39, 40] or to predict whether an error has occurred in a quantum error correction (QEC) code [41]. *Unsupervised learning* refers to ML methods dealing with an unlabeled data set, such as for denoising images [42], or by clustering the data, i.e., sorting it into groups [37]. The field of *reinforcement learning (RL)* is the most general of the three, in the sense that supervised and unsupervised learning tasks can be written as RL tasks. An agent interacts with its environment by receiving rewards and performing actions that can change the state of the system [43]. Applications include playing games like Go [44, 45] or correcting errors in a QEC code [46]. Neural networks (NNs) are a type of function approximator used in all of the three disciplines [37,47]. A particular area of interest in this thesis is the application of ML to quantum mechanical systems [48–50].

The simulation of many quantum mechanical systems is hard for classical computers, in the sense that more than polynomial resources in system size would be needed [51]. Hence, in 1982, R. Feynman proposed the idea of using a fixed quantum system, a *quantum computer (QC)*, to simulate any quantum system [52]. Instead of relying on electrical circuits to perform calculations, QCs use quantum circuits to perform computations [51]. However, quantum mechanical tasks are not the only ones hard to perform on a classical computer

with current algorithms. There are a number of *quantum* algorithms that offer a speed-up over current classical algorithms for specific tasks, given access to a QC [53–55]. This field of research has become increasingly important in recent years, as *Moore's law*, which describes the prediction from 1965 [56] that the computing power doubles every 1.5 to two years at constant cost, is faltering [57]. So, next to improving classical processors, research now also focuses on finding other computational resources, in particular quantum processors. In the last years, significant progress has been made on the experimental realization of QCs [58–63].

Due to the relevance of ML, it is not surprising that several quantum algorithms for ML are proposed [64–66]. So, not only is ML used to analyze quantum mechanical data, but quantum algorithms are used for ML. Quantum machine learning (QML) is the field on the intersection of quantum information (QI) and ML. It is split into four subfields characterized by the type of data (first letter) and algorithm (second letter) used (see, e.g., [66–68]). In the context of QML, the term CC ML describes studying classical data using classical algorithms employing methods such as tensor networks [69] that are inspired by quantum mechanics. CQ ML employs quantum algorithms for studying classical data with the main aim of speeding up classical ML [70–72]. Examining quantum systems using classical algorithms, for example, for quantum error correction (QEC) [73–76] or simulating quantum many-body systems [77], falls into the field of QC ML. When analyzing quantum data, it can be helpful to employ quantum ML algorithms [78–81]. This field is called QQ ML. An example of a quantum neural network (QNN) architecture in the context of QQ ML is a dissipative quantum neural network (DQNN). These networks are able to approximate any map from quantum states to quantum states, called *quantum channels* [82]. In this thesis, we want to focus on using quantum or classical ML algorithms for quantum mechanical or optical problems.

One of the most typical problems in theoretical physics is determining the time evolution of a system given an initial state or values of variables fully categorizing it [83]. A prominent example is the movement of celestial objects, where the state is fully described by the momenta and positions of the celestial objects [84, 85]. One can think of two main variants using SL in such a task: either predicting the future value of a variable based on its past values or predicting the value of a correlated variable using the historical data of the first variable. These time series prediction tasks fit into a broader class of SL problems for which the standard assumption of independent and identically distributed (i.i.d.) data points does not hold. Those data sets in which one data point depends on other data points in the set are called *sequential* [86]. Such tasks can, for example, be found in natural language processing (NLP) [87, 88]. In ML, the standard NN architecture for analyzing sequential data, especially time series, is a recurrent neural network (RNN). In these networks, when inputting data sequentially, the output at time t will not only depend on the input at time t but also the earlier inputs at times $t' < t$ [89].

Quantum processes, for which outputs at time t only depend on inputs at times $t' \leq t$, are referred to as causal quantum automata. These processes can be written as quantum channels with memory, which are similarly constructed from quantum channels [90] like RNNs from NN blocks [89]. A standard example of a quantum channel with memory is the *delay channel*, for which the output at time t is given by the input at time $t - k$ [90].

In order to approximate such causal quantum automata, we propose a type of quantum recurrent neural network (QRNN) structure called dissipative quantum recurrent neural network (DQRNN). Those are based on a certain type of QNN, called DQNN, first presented

in [82]. Some of the total output of the DQNN is used again as part of the total input of the DQNN to form a DQRNN. This process of iterating over an underlying DQNN is similar to quantum channels with memory and classical RNNs. Since DQNNs are able to approximate any quantum channel [82], and causal quantum automata can be written as quantum channels with memory [90], DQRNNs are able to approximate any causal quantum automaton.

Building on this structure, we present quantum and classical training algorithms for both a local and global cost using quantum data as input and output. These algorithms scale with the width of the underlying DQNN, but not its depth. In the global case, the quantum algorithm additionally scales with the number of iterations over the DQNN, and the classical algorithm with the bond dimension of the input and output states. One can either see the classical algorithms as a simulation of training a DQRNN on a QC or as using a tailored network architecture when given classical representations of quantum states. They exhibit a similar structure to the training algorithms of classical NNs, consisting of feed-forward and backpropagation parts. We evaluate the performance of most classical training algorithms numerically on two key examples: the delay channel and the time evolution of a state under a time-dependent Hamiltonian.

Beyond QNNs, ML techniques can also be employed in the control of experimental setups needed for the implementation of QCs. In the realization of many of the current candidates for QCs, complicated laser systems are used [91]. These laser systems regularly have to be aligned and/or controlled. Fibers are a standard tool to guide a laser beam from one place to another in many of those laboratories [91–94]. If a fiber is used after the free propagation of light, the optical path has to be aligned so that the laser beam is guided through the fiber. This is called *fiber coupling*, and the efficiency of this procedure is given by the percentage of light exiting the fiber compared to the power which is put into the fiber.

While RL is the ML technique most suited for control tasks [43], it has mostly been applied to simulated or toy environments [95–99]. RL agents are rarely trained directly in experiments [99] as this induces several challenges, such as time-consuming training, noise, and *partial observability* [97, 98]. The latter means that the state of the system is not fully observable, and only some features of the state can be observed [47]. If those challenges are overcome, RL could be used to automate alignment in experiments, saving time for the human experimenters.

In this thesis, we discuss how an RL agent successfully learns to couple a laser beam into a fiber. It starts from a low coupling efficiency, and uses two mirrors with two motorized axes each to reach a coupling efficiency comparable to human experts. For it to be successful, we had to deal with three main challenges. First, the actuator movements are imprecise, which makes our actions noisy and leads to us not being able to rely fully on the absolute position of the actuators in the reset method. We find that using a reset method that mostly relies on relative positions and training the agent directly in the experiments helps deal with this. Second, each environment step takes about a second in the experiment. This is a hard constraint set by the actuators we used and leads to time-consuming training. We implemented a simplified virtual testbed, not including noise, to test out several different environment designs and algorithms in a short time. As we ultimately train our agents directly in the lab, we had to pay special attention to requiring a limited number of agent-environment interactions. Hence, we focus on algorithms reusing samples from those interactions, like soft actor-critic (SAC) [100, 101], truncated quantile critics (TQC) [102], and twin delayed

deep deterministic policy gradient (TD3) [103]. Third, we built the environment in a way that the agent can only observe the power behind the fiber, not the absolute position of the actuators, as we want to be able to realign the experiment if the optimal positions change. This makes our environment partially observable and underdetermined, as we only have a scalar feedback for four actuators. Using the virtual testbed, we can design a fitting observation.

Outline

As we need basics from both ML and QI, we will introduce them in the first chapters. Depending on their background, the reader can skip one or both of them.

Chapter 2 introduces ML. Unsupervised learning is not needed in the later parts of the thesis, so we focus on SL and RL. In Section 2.1, we establish a SL framework and discuss (recurrent) neural networks and their training algorithms which we need for two reasons. First, introducing the classical variant facilitates a clearer understanding of DQRNNs. Second, NNs are used in most modern RL algorithms, which we employ to fiber couple. Those algorithms will be discussed in Section 2.2 after defining the RL framework.

We lay the foundations needed from QI in Chapter 3. Especially sections 3.1-3.5, 3.7, and 3.10, where we recap basic definitions, are aimed at people not familiar with QI. To better understand the general structure of DQRNNs, we introduce quantum channels with memory. For SL of quantum states, we need distance measures and a way to sample quantum states. Additionally, we establish the notion of tensor networks (TNs) to deal with entangled states more efficiently. Lastly, we explore QML, particularly DQNNs.

In Chapter 4, we propose a recurrent version of DQNNs as presented in [104]. After showing that they can be used to approximate any causal quantum automaton, we compare them to other QRNN architectures. We then define loss functions for different kinds of training data, for which we develop training algorithms. Using classical training algorithms, we present numerical results. The derivation of the classical training algorithms can be found in Appendix A.

We discuss using RL for fiber coupling in Chapter 5 as presented in [105]. After introducing the task of fiber coupling from an RL perspective, we design the RL environment. Using a virtual testbed, we tune environment parameters and select training algorithms. We then finally present results from a fiber coupling experiment in a laboratory. Further results can be found in Appendix B.

The thesis finishes with a conclusion and outlook in Chpter 6.

2

Classical Machine Learning

Consider the task of developing a computer program that distinguishes handwritten digits. It is hard to determine which pixels exactly have to be of a certain color, especially if multiple people write the handwritten digits not perfectly. Still, for handwritten digits, we can at least look at the picture and know the right answer. That makes it possible to label thousands of pictures with the correct answer. For other tasks like learning how to play games such as Go or Football, it is not so clear what the label would be. Even as a human expert, determining how to act in every situation is difficult. So, how do humans learn how to evaluate such complex situations? Instead of being exactly instructed on what to do in every possible situation, we get to see a lot of examples, e.g., handwritten digits, or repeat games many times and maybe observe other people's gameplay to determine the best strategies. As humans, we learn from examples and interactions with our environment. The field of machine learning (ML) imitates this kind of behavior - taking a lot of examples, called *experience*, and using them to predict labels of later seen inputs, cluster data, i.e., sort data into groups, or improve performance in a game [37, 38].

We can sort most ML algorithms into three main categories by their human supervision requirements: supervised, unsupervised, and reinforcement learning. In *supervised learning (SL)*, the *learner* or *agent* is given a labeled data set or, more generally, an input-output set and must find a function fitting the data. Then, the agent predicts the output/label of future instances. Examples of this are classification tasks, like recognizing handwritten digits, and regression tasks, like predicting the measured power in an optical experiment. In *unsupervised learning*, the agent receives an unlabeled data set. Here, tasks include clustering or estimating the density distribution of data. In *reinforcement learning (RL)*, the agent can interact with a given environment, e.g., a game, by performing actions, observing the state of the environment, and being rewarded for these actions [37].

In this thesis, we focus on supervised and reinforcement learning, so in this chapter, we highlight the parts of these fields we later need and disregard unsupervised learning at this time. In Section 2.1, we introduce SL and then provide an overview of RL in Section 2.2.

2.1 Supervised Learning

In supervised learning (SL), we are given input-output pairs and want to find a map from the inputs to the outputs fitting the data as best as possible. One example of this is the

recognition of handwritten digits. Here, inputs consist of pictures of handwritten digits, i.e., matrices of pixels, and the output consists of the corresponding digits. We are given a number of examples of input-output pairs, 60 000 if we are using the standard data set MNIST [106]. Then, we fit function approximators like neural networks (NNs) to this data and try to predict the outputs of unseen inputs. Employing these function approximations, we can then automate tasks like the recognition of handwritten digits [107].

This section is structured as follows. First, we give a formal introduction to the SL framework in Section 2.1.1. Then, we consider different losses to assess the performance of a given function approximation in Section 2.1.2. These definitions reveal different kinds of errors, which we discuss in Section 2.1.3. We then discuss today's most common function approximator, NNs in Section 2.1.4. This includes the definition of perceptrons and feed-forward NNs, showing that they are universal approximators, their training algorithm, and the definition of recurrent NNs. In Section 2.1.5, we discuss hyperparameters, i.e., parameters that have to be chosen before training a function approximator. Furthermore, we discuss how to handle data sets and discuss the generalization behavior of approximators in Section 2.1.6.

2.1.1 Formal Framework

We formalize the SL framework by adapting from [38, 89, 108]. Let $\mathcal{X}$ and $\mathcal{Y}$ be measurable spaces, where $\mathcal{X}$, also known as *domain, feature* or *instance* set, is associated with the input system and $\mathcal{Y}$, also known as *label* set, with the output system. A SL algorithm $\mathcal{A}$ uses a data set $S = (x_i, y_i)_{i=1,\dots,N} \in (\mathcal{X} \times \mathcal{Y})^N$ as input and returns a hypothesis $h_S : \mathcal{X} \to \mathcal{Y}$. For now, we assume that the data in S is independent and identically distributed (i.i.d.) w.r.t. a probability measure $\mathcal{P}$ on $\mathcal{X} \times \mathcal{Y}$. We call the set of functions available to the learning algorithm $\mathcal{F} \subseteq \mathcal{M}(\mathcal{X}, \mathcal{Y})$ the *hypothesis class*, where $\mathcal{M}(\mathcal{X}, \mathcal{Y})$ is the set of measurable functions from $\mathcal{X}$ to $\mathcal{Y}$. With every SL task, we associate a *loss* function $l : \mathcal{M}(\mathcal{X}, \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to \mathbb{R}_+$ that can be used as a measure of performance for a hypothesis $h \in \mathcal{F}$. We write $\mathbb{P}_A[B]$ for the probability of an event B, given the constraint A and $\mathbb{E}_A[X]$ for the expectation value of a stochastic variable X. Furthermore, we write $a \sim \mathcal{P}$ to express that a is sampled from $\mathcal{X} \times \mathcal{Y}$ w.r.t. $\mathcal{P}$. We define

$$R_{\mathcal{P}}(h) = \mathbb{E}_{(x,y)\sim\mathcal{P}}[l(h,(x,y))] = \int_{\mathcal{X}\times\mathcal{Y}} \mathrm{d}x\, \mathrm{d}y\; l(h,(x,y)) \tag{2.1.1}$$

to be the *risk* or *true error* of h and, given a set $S = ((x_i, y_i))_{i=1,\dots,N} \sim \mathcal{P}^N$, we define

$$\hat{R}_S(h) := C_S(h) := \frac{1}{N} \sum_{i=1}^{N} l(h,(x_i,y_i)) \tag{2.1.2}$$

as the *empirical risk* or *cost*, which is often used as an approximation for the risk. In short, we sometimes write

$$c_i = l(h,(x_i,y_i)).$$

Given $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{P}$, we say a hypothesis $h^* \in \mathcal{M}(\mathcal{X}, \mathcal{Y})$ is *optimal* iff it minimizes the risk. Its risk is also called the *Bayes risk*

$$R^*_{\mathcal{P}} = R_{\mathcal{P}}(h^*) = \min_{h\in\mathcal{M}(\mathcal{X},\mathcal{Y})} R_{\mathcal{P}}(h). \tag{2.1.3}$$

A learning algorithm $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^N \to \mathcal{F}$ uses a data set $S = (x_i, y_i)_{i=1,\dots,N} \sim \mathcal{P}^N$ as input and returns a hypothesis $h_S \in \mathcal{F}$. The goal of the learning algorithm is to minimize $R_{\mathcal{P}}(h_S)$

by finding h^*. To do so, it only has access to S, not $\mathcal{P}$, so $R_{\mathcal{P}}(h_S)$ cannot be directly evaluated. Instead, it has to rely on $\hat{R}_S$. Additionally, the optimal hypothesis h^* could even be unreachable to the learning algorithm as it could not be in the hypothesis class $\mathcal{F}$ chosen by us [38, 89, 108].

2.1.2 Loss Functions

Common loss functions for *classification tasks*, i.e. SL tasks, where the label set $\mathcal{Y}$ is finite, are the the 0-1-*loss* [38, 108]

$$l_{0-1}(h,(x,y)) = 1 - \delta_{h(x),y},$$

where δ is the Kronecker delta, or the *relative entropy* or *Kullback–Leibler (KL) divergence* [109]

$$l_{\mathrm{KL}}(h,(x,p(y))) = \sum_{y\in\mathcal{Y}} p(y)\ln\frac{p(y)}{h(x)}$$

if we focus on learning a target probability distribution p over the target set [109]. Other choices include losses usually used for *regression*, which are SL tasks with $\mathcal{Y} = \mathbb{R}$. Given a metric space $\mathcal{Y}$ with metric d, two very common loss functions are the *quadratic loss*

$$l_{\mathrm{sq}}(h,(x,y)) = d(y,h(x))^2$$

or the *absolute error loss*

$$l_{\mathrm{ae}}(h,(x,y)) = d(y,h(x)).$$

For $\mathcal{Y} = \mathbb{R}$ with the usual metric, the associated costs are called the *mean squared error (MSE)* and the *mean absolute error (MAE)*, respectively [38, 108, 110]. In the MSE, outliers get more weight compared to the MAE, which can make it less robust. On the other hand, the MAE is not differentiable at $y = h(x)$. This can make it hard to use for optimization. By combining the two, using the *Huber loss*

$$l_\kappa^H(h,(x,y)) = \begin{cases} \frac{1}{2}d(y,h(x))^2, & \text{if} \quad d(y,h(x)) \le \kappa \\ \kappa(d(y,h(x)) - \frac{1}{2}\kappa), & \text{else} \end{cases}$$

where $\kappa \in \mathbb{R}_{>0}$, we can use the advantages of both [111, 112].

2.1.3 Error Types

As discussed before, the learner's task is to get the optimal hypothesis $h^* \in \mathcal{M}(\mathcal{X},\mathcal{Y})$ for the underlying probability measure $\mathcal{P}$ on $\mathcal{X}\times\mathcal{Y}$ using a given loss function $l : \mathcal{M}(\mathcal{X},\mathcal{Y}) \times (\mathcal{X}\times\mathcal{Y}) \to \mathbb{R}_+$ as its measure of performance. It does not have access to $\mathcal{P}$ and usually not all of $\mathcal{M}(\mathcal{X},\mathcal{Y})$ to do so, but only $S = (x_i,y_i)_{i=1,\dots,N} \sim \mathcal{P}^N$ and functions in the hypothesis space $\mathcal{F} \subseteq \mathcal{M}(\mathcal{X},\mathcal{Y})$. We write $\hat{h}^*_{S,\mathcal{F}}$ for the hypothesis minimizing the empirical risk on $\mathcal{F}$ and $h^*_{\mathcal{F},\mathcal{P}}$ for the hypothesis minimizing the risk on $\mathcal{F}$. The error ϵ, i.e. the difference between the risk $R_{\mathcal{P}}(h)$ for a picked hypothesis h and $R^*_{\mathcal{P}}$, can then be decomposed into three parts

$$\begin{aligned}\epsilon &= R_{\mathcal{P}}(h) - R^*_{\mathcal{P}} \\ &= R_{\mathcal{P}}(h) - \hat{R}_S(h) + \hat{R}_S(h) - \hat{R}_S(h^*_{\mathcal{F},\mathcal{P}}) + \hat{R}_S(h^*_{\mathcal{F},\mathcal{P}}) - R_{\mathcal{P}}(h^*_{\mathcal{F},\mathcal{P}}) + R_{\mathcal{P}}(h^*_{\mathcal{F},\mathcal{P}}) - R^*_{\mathcal{P}} \\ &\le \epsilon^{\mathrm{gen}}_{\mathcal{F},S,\mathcal{P}} + \epsilon^{\mathrm{opt}}_{\mathcal{F},S,\mathcal{P}}(h) + \epsilon^{\mathrm{gen}}_{\mathcal{F},S,\mathcal{P}} + \epsilon^{\mathrm{approx}}_{\mathcal{F},\mathcal{P}} = \epsilon^{\mathrm{opt}}_{\mathcal{F},S,\mathcal{P}}(h) + 2\epsilon^{\mathrm{gen}}_{\mathcal{F},S,\mathcal{P}} + \epsilon^{\mathrm{approx}}_{\mathcal{F},\mathcal{P}}\end{aligned}$$

with the *approximation error*

$$\epsilon^{\text{approx}}_{\mathcal{F},\mathcal{P}} = R_{\mathcal{P}}(h^*_{\mathcal{F},\mathcal{P}}) - R^*_{\mathcal{P}},$$

the *optimization error*

$$\epsilon^{\text{opt}}_{\mathcal{F},S,\mathcal{P}}(h) = \hat{R}_S(h) - \hat{R}_S(h^*_{\mathcal{F},\mathcal{P}}),$$

and the *generalization error*

$$\epsilon^{\text{gen}}_{\mathcal{F},S,\mathcal{P}} = \sup_{h\in\mathcal{F}} |R_{\mathcal{P}}(h) - \hat{R}_S(h)|.$$

The approximation error serves as an indicator of how well the hypothesis class $\mathcal{F}$ can approximate the optimal hypothesis $h^*_{\mathcal{F},\mathcal{P}}$. The optimization error quantifies how good the hypothesis h found by the optimization process is compared to ideal empirical risk minimization. The generalization error serves as an indicator of the quality of the approximation of the true risk by the empirical risk on the hypothesis class, which in turn can indicate how well a trained hypothesis can generalize to unseen data. When minimizing the approximation error by choosing a larger hypothesis class, we usually get a higher generalization error. This is known as the *bias-variance trade-off*, as a general hypothesis class usually is not very biased but the trained hypotheses vary a lot, and a very simple hypothesis class is very biased to its definition but the trained hypotheses are nearly the same. We usually want to balance both extremes [89, 108].

2.1.4 Neural Networks

A commonly used hypothesis class, inspired by models of the human brain, are neural networks (NNs). The way humans and animals process information is very different from how conventional digital computers do so. Depending on the task, this leads to an advantage or disadvantage of conventional digital computers compared to humans. In highly simplified terms, the brain is given a structure built from connected neurons and learns to establish some connections via experience and interaction with the environment. In 1943, McCulloch and Pitts published a mathematical formalization of a model of human neurons [113]. Based on their results, *(artificial) neural networks (NNs)* were built to model human information processing. NNs should process different information paralleled and acquire knowledge from an environment via learning, and its knowledge should be stored in the connection strengths (weights) between neurons [114].

Nowadays, (deep) NNs are the outstanding technique, not only in SL but in ML in general [89]. They are used, e.g., for the classification of images [115, 116], natural language processing (famously in ChatGPT) [7, 117], for playing games (in the context of RL) [44, 45, 118–120], and in various fields of science [41, 121, 122].

Perceptrons

Based on the work by McCulloch and Pitts, Rosenblatt developed a certain type of artificial neuron [123] starting in the 50s and 60s. His model is called the *perceptron, Rosenblatt perceptron* or *Rosenblatt neuron* depending on how narrow the perceptron is defined. The model is the following:

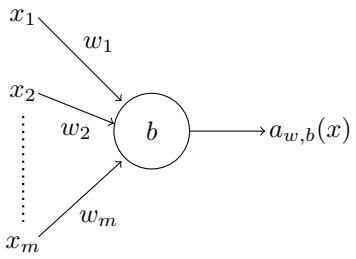


Figure 2.1: **Perceptron** with inputs $x = (x_i)_{i=1,\dots,m} \in \{0,1\}^m$, a bias $b \in \mathbb{R}$, weights $w = (w_i)_{i=1,\dots,m} \in \mathbb{R}^m$ and an output $a_{w,b}(x) \in \{0,1\}$. The map consists of weighing the inputs, summing them up, adding the bias, and then using the step function to calculate the output.

Given $m \in \mathbb{N}$ inputs $x = (x_i)_{i=1,\dots,m} \in \{0,1\}^m$, *weights* $w = (w_i)_{i=1,\dots,m} \in \mathbb{R}^m$ and a *bias* b, the output is given by

$$a_{w,b}(x) = \theta\left(\sum_{i=1}^{m} w_i x_i + b\right) = \begin{cases} 1, & \sum_{i=1}^{m} w_i x_i + b \geq 0 \\ 0, & \sum_{i=1}^{m} w_i x_i + b < 0 \end{cases} \tag{2.1.4}$$

where θ is the *step* or *Heaviside function*. The perceptron is depicted as shown in Figure 2.1 [107, 114].

This output rule has one main problem that makes the learning process hard: the Heaviside function is not continuous, i.e., a small change in the weights or the bias can lead to a big change in the output. This is why other functions are used instead of the Heaviside function in most cases today [107]. In the following, we adopt the convention of also calling these neurons perceptrons. In general, we have

$$a_{w,b}(x) = \varphi\left(\sum_{i=1}^{m} w_i x_i + b\right) = \varphi(z) \tag{2.1.5}$$

where $\varphi : \mathbb{R} \to \mathbb{R}$ is called the *activation function* and $z = \sum_{i=1}^{m} w_i x_i + b$. As explained in more detail later, we usually choose φ to be non-linear and continuous, where we need the non-linearity for the universality of neural networks and the continuity for most learning algorithms. Examples of activation functions with a similar global form (compared to the step function) are the hyperbolic tangent (tanh) [89] or the *sigmoid* or *logistic* function

$$\varphi_{\text{sigmoid},\alpha}(z) = \frac{1}{1 + \mathrm{e}^{-\alpha z}}.$$

Neurons with this activation function are often called *sigmoid neurons* [114]. These functions are well suited if we want our outputs to be confined to an interval, but they have the disadvantage of *vanishing gradients* for large input variables, i.e., $\phi'(z) \to 0$ for $|z| \gg 1$. If one wants to deal with this issue and does not need the outputs to lie in a given interval, piece-wise linear functions can be a good choice. An example of one such activation function is the *rectified linear* one $\varphi_{\text{ReLU}}(z) = \max(0, z)$, whose neurons are often called *rectified linear unit (ReLU)* [108].

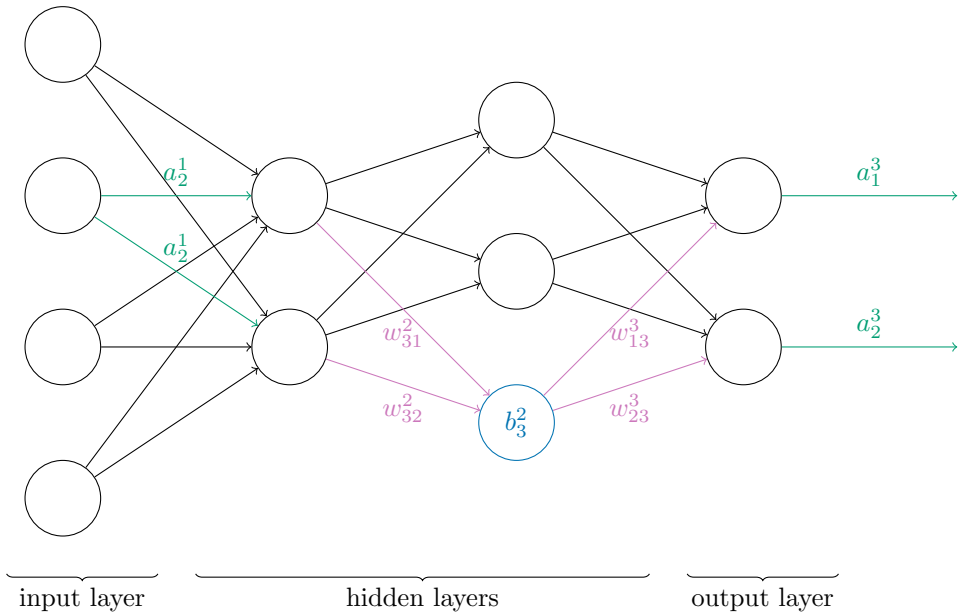


Figure 2.2: **Classical feed-forward neural network.** A ff NN consists of connected perceptrons. With neuron j in layer l , we associate a bias b_j^l, and a weight w_{jk}^l with every connection to a node k in layer $l-1$. Neuron j in layer l outputs a_j^l, where $a_j^l = \varphi^l\left(\sum_{k=1}^{m_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l\right)$ for $l > 1$ using activation function φ^l.

Feed-Forward Neural Networks

These perceptrons are then connected in layers to get a basic (or *vanilla*) *neural network (NN)*. The first layer is called the *input layer* (layer 0). These neurons do not have any input, and they output the input of the NN. The last layer is called the *output layer*, and its output is the output of the NN. The layers in between are called *hidden layers*. A connection between two neurons means that the output of one neuron is used as input for the other neuron.[1] For now, let us assume that information flows from layer $l-1$ to layer l (or left to right). NNs with this property are called *feed-forward (ff)* NNs. If all neurons in each layer of a NN are connected to all neurons in the next layer, we call it *fully connected (fc)* [89,107].

We use the following notation: Let φ^l be the activation function and m_l be the number of neurons in layer $l \in \{0, ..., L+1\}$. The upper index of a variable always signifies the layer, and the lower index is the position of the perceptron in that layer (counted from the top). For perceptron $j \in \{1, ..., m_l\}$ in layer l, we write a_j^l for it's output, b_j^l for it's bias, and w_{jk}^l for it's weight associated with the connection to neuron k in layer $l-1$. This means we have

$$a_j^l = \varphi^l\left(\sum_{k=1}^{m_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l\right).$$

[1] In cases where it is unclear which neuron is the output and which is the input neuron, we will indicate the direction of the information flow with an arrow.

We can also vectorize this notation by writing

$$a^l = \begin{pmatrix} a_1^l \\ \vdots \\ a_{m_l}^l \end{pmatrix}, \quad b^l = \begin{pmatrix} b_1^l \\ \vdots \\ b_{m_l}^l \end{pmatrix}, \quad W^l = \begin{pmatrix} w_{11}^l & \dots & w_{1m_{l-1}}^l \\ \vdots & \ddots & \vdots \\ w_{m_l1}^l & \dots & w_{m_lm_{l-1}}^l \end{pmatrix}. \tag{2.1.6}$$

The forward process then becomes

$$a^l = \varphi^l\left(W^l a^{l-1} + b^l\right) = \varphi^l\left(z^l\right), \quad z^l = W^l a^{l-1} + b^l \tag{2.1.7}$$

where φ^l acts component-wise. We can formalize NNs with the following definition [107].

Definition 2.1 (Fully connected feed-forward neural network (adapted from [89]))**.** *Let $L \in \mathbb{N}$, $m = (m_l)_{l=0,\dots,L+1} \in \mathbb{N}^{L+2}$, $\varphi = (\varphi^l)_{l=0,\dots,L+1}$ where $\varphi^l : \mathbb{R} \to \mathbb{R}$ for $l \in 1, \dots, L+1$. On vectors the activation functions $(\varphi^l)_{l=0,\dots,L+1}$ are applied component-wise. An architecture (m, φ) defines a function class of* fully connected feed-forward neural networks *given by*

$$\mathcal{F}_{(m,\varphi)} = \Big\{ h_{W,b} : \mathbb{R}^{m_0} \to \mathbb{R}^{m_{L+1}} \Big| h_{W.b}\left(a^0\right) = a_{W.b}^{L+1},\ a_{W.b}^l = \varphi^l\left(W^l a_{W,b}^{l-1} + b^l\right), \\ W^l \in \mathbb{R}^{m_l \times m_{l-1}}, b^l \in \mathbb{R}^{m_l},\ l = 1, \dots, L+1, \\ W = (W^l)_{l=1,\dots,L+1},\ b = (b^l)_{l=1,\dots,L+1} \Big\}.$$

The width *of the network is $\max_{l \in 0,\dots,L+1} m_l$, and it's* depth *is L. We say the network is* shallow *if $L = 2$ and* deep *if $L > 2$.*

Universality

Neural Networks are said to be *universal approximators*, i.e., given a high enough number of neurons, they can approximate any continuous function to any given degree. The following theorem is an example of a universal approximation theorem, showing that when using any non-constant, bounded, monotonously increasing, and continuous function on $[0, 1]$ as activation function, we can approximate every continuous function by a neural network with 1 hidden layer and a linear output layer.

Theorem 2.1 (Universal Approximation Theorem [114])**.** *Let $C(K)$ denote the continuous functions from a set K to $\mathbb{R}$. Let $\phi \in C(\mathbb{R})$ be non-constant, bounded and monotonously increasing, $m_0 \in \mathbb{N}$. For all $f \in C([0,1]^{m_0})$ and $\epsilon > 0$, there is $m_1 \in \mathbb{N}$, α, $\beta \in \mathbb{R}^{m_1}$ and $w \in \mathbb{R}^{m_1} \times \mathbb{R}^{m_0}$ such that*

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon$$

with

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \phi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right).$$

The presented theorem considers the bounded depth (number of layers) and arbitrary-width (maximum number of neurons per layer) case. Other theorems for this case can be found in [124–127]. But there are also a number of theorems for the arbitrary depth and bounded width case [128–130] and even for the bounded depth and width case [131, 132].

Gradient Descent

Often, when trying to minimize the cost given a certain NN architecture, the learner has to do so w.r.t. a large number of parameters. Hence, it usually cannot just set the gradient to zero as we would do for analytically finding the extrema since it takes too long to solve the equation. Instead, one usually employs a kind of iterative algorithm. Variants of *gradient descent* are the standard learning algorithms for NNs [107]. But it is, more generally, an algorithm for numerically finding local minima of differentiable functions and, in the context of convex unconstrained minimization, even the global minimum [133].

First let us define the problem: Given a function $f : \mathbb{R}^n \to \mathbb{R}$ with input $v \in \mathbb{R}^n$, we want to find

$$v^* = \arg\min_{v \in \mathbb{R}^n} f(v).$$

An iterative algorithm is an algorithm that iteratively outputs a sequence of points $v^{(k)}$, i.e., $v^{(k)} = g\left(\left(v^{(k-i)}\right)_{i=1,\dots,m}\right)$ for some function $g : \mathbb{R}^{nm} \to \mathbb{R}^n$ and some $m \in \mathbb{N}$. One easy example for an iterative algorithm is to determine a *step direction* $\Delta v^{(k)}$, a *step size* or, in the context of ML, *learning rate* η and choose $v^{(k+1)} = v^{(k)} + \eta \Delta v^{(k)}$. Algorithms like this solving a minimization problem are called *descent* algorithms. The gradient is a common choice for the step direction, i.e., we take $\Delta v^{(k)} = -\nabla f\left(v^{(k)}\right)$. This algorithm is called *gradient descent* [133].
Translated to NNs, this means that the weights and biases are updated in each step according to

$$w^l_{jk} \to w^l_{jk} - \eta \frac{\partial C_S}{\partial w^l_{jk}}, \quad b^l_j \to b^l_j - \eta \frac{\partial C_S}{\partial b^l_j}.$$

In practice, if we are given a big training set of ten- or hundred-thousands of training samples, it can take long to calculate the gradient of the cost, as defined in Equation (2.1.2),

$$\nabla_{w,b} C_S(w,b) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{w,b} l\left(h_{w,b}, (x_i, y_i)\right).$$

A more efficient method, called *stochastic gradient decent*, divides the training set randomly in sets $S = (S_1, ..., S_m)$, called *mini-batches*, of size $n \in \mathbb{N}$ each, called *mini-batch size*. If n is big enough, it is

$$\nabla_{w,b} C_S(w,b) \approx \nabla_{w,b} C_{S_i}(w,b).$$

The training is then divided in *epochs*. Each epoch consists of steps in which the weights and biases are updated according to the i^{th} mini-batch. In doing so, fewer terms of the cost have to be computed per update. The epoch is finished after a step with each of the mini-batches was performed [107].

Backpropagation

As NNs can have a high number of parameters, we need a more efficient algorithm than brute-force computing all derivatives. Such a method is *backpropagation*. It was first introduced in the 70s [134] but only fully appreciated after a 1986 paper [135]. When changing the variable z^l_j slightly by Δz^l_j, the cost will approximately change as $\frac{\partial C_S}{\partial z^l_j} \Delta z^l_j$. Hence, we introduce

$$d^l_j = \frac{\partial C_S}{\partial z^l_j}$$

as the *error* in neuron j in layer l. We can easily derive the error in the last layer using the chain rule as[2]

$$d_j^{L+1} = \frac{\partial C_S}{\partial z_j^{L+1}} = \frac{\partial C_S}{\partial a_j^{L+1}} \frac{\partial a_j^{L+1}}{\partial z_j^{L+1}} = \frac{\partial C_S}{\partial a_j^{L+1}} \varphi' \left(z_j^{L+1} \right) .$$

Using the *Hadamard product* "$\circ$," i.e., the elementwise product $(s \circ t)_j = s_j t_j$, we can write this as

$$d^{L+1} = \nabla_{a^{L+1}} C_S \circ \varphi' \left(z^{L+1} \right) . \tag{2.1.8}$$

Again, using the chain rule, we can trace the error back through the network via

$$d^l = \left(\left(W^{l+1} \right)^T d^{l+1} \right) \circ \varphi' \left(z^l \right) \tag{2.1.9}$$

for $l = 0, ..., L$ where A^T denotes the transpose of a matrix A. This is why the algorithm is called the backpropagation algorithm. We then can write the partial derivatives over the weights and biases as

$$\frac{\partial C_S}{\partial b_j^l} = d_j^l, \quad \frac{\partial C_S}{\partial w_{jk}^l} = d_j^l a_k^{l-1} \tag{2.1.10}$$

for all $l = 1, ..., L+1$, $j = 1, ..., m_l$ [107].

Recurrent Neural Networks

Until now, we looked at ff NNs, that is, NNs without loops, where information flows from one layer to the next. In contrast, NNs with loops of any kind are called *recurrent neural networks (RNNs)*. In RNNs [89], the input of a layer l can also depend on the outputs of layers l to $L+1$ and not only 0 to $l-1$ (see Figure 2.3 (a)). This means that computations of the internal state of the NN $\{a^l\}_{l=0,...,L+1}$ cannot simply be performed layer-by-layer. Instead, we introduce a time parameter t. In each timestep $t \to t+1$, all possible computations to the current internal or *hidden* state h_t of the RNN are applied to get a new hidden state h_{t+1}. We can make this clear by unfolding the RNN as shown in Figure 2.3 (b). If our input data is a sequence, where we input an element a_t^{in} in each time step, the output at time t can depend on all inputs $a_{t'}^{\text{in}}$ for $t' \leq t$. We can thus say that it exhibits a *memory*. Because of its definition, RNNs are especially useful for learning sequential data, like time evolution [136], speech recognition [137], the prediction of electric power demand [138], and machine translation [139]. Particularly, in natural language processing (NLP) RNNs were widely used [140].

By assigning a hidden state of the RNN, we can simplify Figure 2.3 to Figure 2.4 [141]. For vanilla RNNs, all of the cells are basic ff NNs. Essentially, an unfolded RNN can be seen as a very deep ff NN sharing a lot of parameters between layers. As such, a learning algorithm often used is *backpropagation through time* [142]. On the downside, this also means that vanilla RNNs, like the ones discussed until now, share a major problem with deep ff NNs: The gradients are often either exploding or vanishing, i.e., get extremely large or small with rising depth of the NN, which makes such NNs hard to train. This is why special structures, like *long short-term memory (LSTM) cells* [143] or *gated recurrent units (GRUs)* [144], were introduced [89]. The two most used RNN structures today are GRUs and LSTM cells. Instead of the basic ff NNs, the cells in Figure 2.4 are then replaced by LSTM cells or GRUs. As we will only explore vanilla quantum recurrent neural networks later on, we will not further discuss these more advanced structures.

[2] Note that we do not use Einstein's sum convention here.

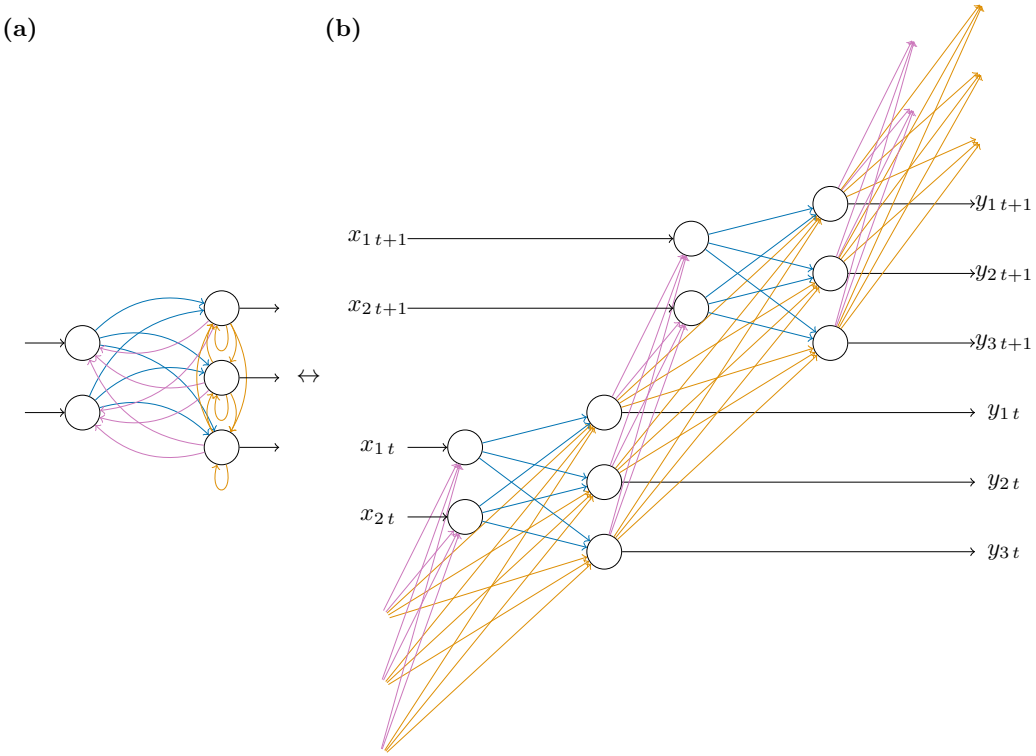


Figure 2.3: **Classical fully connected recurrent neural network.** Panel (a) shows the compressed version of a fc RNN with two input and three output neurons and no hidden layers. Panel (b) shows the same RNN in its unfolded version. The colors indicate the different weight matrices: blue for the weights between the 0^{th} and the 1^{st} layer, purple for the ones between the 1^{st} and 0^{th} layer and orange for the ones between the 1^{st} and 1^{st} layer.

2.1.5 Hyperparameters

In the last section, we saw that there are some parameters, like the learning rate or NN architecture, one has to choose before training the NN. Parameters like this that have to be chosen and are not determined by the learning algorithm are called *hyperparameters* [145]. For the rest of this section, we adapt from [107].

In general, these parameters include the loss function and the encoding of the input and output on which we train the hypothesis. In some cases, these are very clear, but in other cases, that is not the case. For example, when training a NN to recognize handwritten digits, one can either choose the outputs to be natural numbers from 0 to 9 and the loss to be the $0-1-$loss, or the outputs to be probability distributions over those numbers and the loss to be the quadratic loss or the cross-entropy.

When using (stochastic) gradient descent, the learning rate η and the number of steps or epochs used are standard hyperparameters. If the learning rate is chosen too high, the optimizer usually just jumps over the optimum in some steps, which leads to oscillations in the cost during training. On the other hand, choosing it too low leads to the training

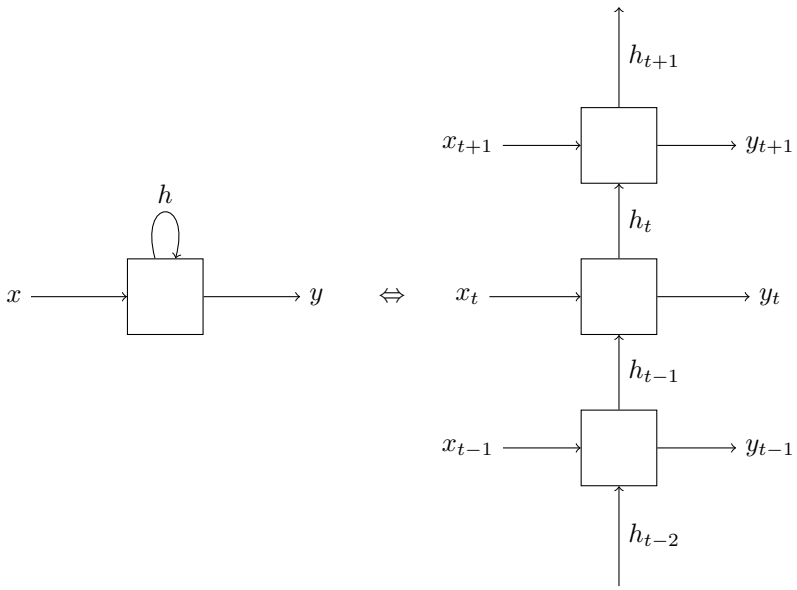


Figure 2.4: **Classical recurrent neural network structure.** General structure of RNNs. The left-hand side shows the compressed version and the right-hand side shows the same unfolded version. For vanilla RNNs, the squares are simple NNs, but they can also be more complicated, e.g. LSTM cells or GRUs.

being very time-consuming. This sometimes makes it beneficial to adjust the learning rate gradually during training. Furthermore, if we choose the number of epochs or training steps to be too low, we will interrupt training before it is finished. However, choosing a too high number of steps or epochs can lead to *overfitting*, i.e., the hypothesis better fits the training data but does not generalize well to unseen data. This issue can be overcome by implementing a termination condition known as *early stopping*, i.e., interrupting the training when the cost on a data set not used for training (the validation set, as discussed in Section 2.1.6) saturates. Of course, the learning rate and number of training steps or epochs are related: the lower the learning rate is, the higher the number of steps or epochs has to be, as training, generally, takes more time with a lower learning rate (if we do not change any other parameters).

More specifically, when using stochastic gradient descent, the mini-batch size is a hyper-parameter. If we choose it too big, the updates of the weights and biases become too time-consuming. In contrast, if we make it too small and use efficient methods such as specialized libraries for matrix multiplication, which we usually do, we will not take enough advantage of them. When using a NN as a function approximator, its architecture, i.e., its width, depth, and activation functions, and the initialization of weights and biases are hyperparameters.

Most people use some kind of heuristics like the ones presented above for choosing the hyperparameters. Still, the coupling between the different parameters can make the choice difficult. One strategy is to start with simpler data, e.g., only 0's and 1's for handwritten digit recognition or less noisy data, and a simple NN architecture - first, we want to learn

anything. Then, we optimize the hyperparameters for this easier case. We then make the task gradually more difficult and optimize the other hyperparameters in each step.

Of course, there are also automated techniques. The easiest is a simple grid search, i.e., an algorithm searches through the hyperparameter space by trying out parameters in a kind of grid, always raising them by a specific amount. But there are also more sophisticated methods, like the Bayesian approach. However, usually, the optimizing process is never really finished: There are almost always better hyperparameter combinations to find. To determine when we are finished, we need to weigh the potential better performance of a model against the resources needed for further optimization.

2.1.6 Generalisation and Data Sets

As a measure of performance for a hypothesis, we are often not only interested in its cost but also in how well the hypothesis works on unseen examples compared to already seen ones, which provides an estimate for the generalization error. Given a hypothesis class $\mathcal{F}$, a data set S and a learning algorithm $\mathcal{A}_v : (\mathcal{X} \times \mathcal{Y})^N \to \mathcal{F}$ depending on some hyperparameters v, we often split the data set into three different data sets: The training set, the validation set and the test set. Here, the *training set* is the set used as input by the learning algorithm, i.e., the set used by the learning algorithm to fit the parameters of the classifier. In the case of neural networks, this would be the set on which the weights and biases are learned using backpropagation. The *validation set* is the set we use to tune the hyperparameters v of the learning algorithm. In the case of NNs, as discussed earlier, this would, e.g., be the selection of the architecture, the learning rate, or the mini-batch size. The *test set* is only used once in the end to measure the performance of a fully specified hypothesis. We do this so we get an unbiased measure of how well our hypothesis generalizes. In total, we would use several different learning algorithms $\mathcal{A}_{v_1}, ..., \mathcal{A}_{v_m}$ on the training set leaving us with different hypotheses $h_{v_1,\theta^*}, ..., h_{v_m,\theta^*}$, then choose the hypothesis h_{v^*,θ^*} with the best performance on the validation set and then evaluate the performance of h_{v^*,θ^*} on the test set to get a final performance measure. After using the test set, the hypothesis or learning algorithm should not be updated anymore [146].

If the data set is small, we sometimes do not have enough data points for a separate validation set. We then, e.g., can use a method called cross-validation or other information criteria further explained in [37].

Now, let us consider reinforcement learning, which is more general but still uses a lot of concepts from SL, as we will see later.

2.2 Reinforcement Learning

Currently, reinforcement learning (RL) is the ML form closest to the way humans and animals learn. In RL, we are given an agent in an environment that can interact with this environment. The agent could, e.g., be a chess player, and the environment could be the chess board with all the figures on top of it and the opposing chess player. At a time t, the environment is in a certain state; in our example, that could be the position of the chess pieces on the board. The agent can perform actions, e.g., moving a chess piece, which can change the state of the environment. Sometimes, the agent receives a reward from the environment, e.g., a $+1$ for winning and a -1 for losing. The agent then wants to maximize his received cumulative reward, called *return*. This back-and-forth between the agent and

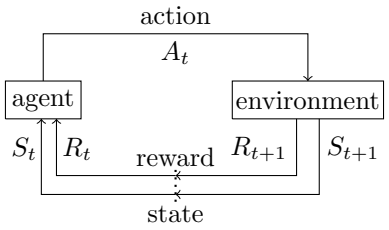


Figure 2.5: **Reinforcement Learning Framework.** Given the environment is in state S_t, the agent can choose to perform the action A_t. The state afterwards is S_{t+1} and the agent receives the reward R_t. This back-and-forth process is repeated in a loop.

the environment is depicted in Figure 2.5. The setup shares many similarities with the field of optimal control, in which a controller is designed to maximize or minimize a measure of the behavior of a dynamical system controlled by the controller. In fact, many of the earlier developments in the field of RL stem from the field of optimal control, e.g., Markov decision processes (MDPs) and some of the earlier algorithms like dynamic programming for solving those. In a control setting, the agent would be called the *controller*, the environment would be the *controlled system* or *plant*, and the action would be called the *control signal*. In the remainder of this thesis, we will stick to the RL terms [43].

Especially in games, RL was used with a lot of success: Famously, in 2016, as the first program AlphaGo beat the world champion Lee Sedol in Go 4-1, although it was seen as the hardest classical game for AI [44, 45]. AlphaGo used a combination of SL from human expert movements and RL from self-play; the later published AlphaGo Zero relied solely on RL for reaching superhuman play [45]. Even more general, AlphaZero is not specialized in only one game but was shown to beat state-of-the-art programs specializing in chess, shogi and Go [147]. But RL was used in not only classical but also computer games: Already in 2015, an RL agent reached superhuman levels in a number of Atari Games [148]. Two years later, AlphaStar [120] defeated the world's strongest player in Starcraft II, and another year later, OpenAI Five [119] defeated the world champions in Dota 2 [47].
Apart from games, AlphaZero was successful in a completely different area: AlphaTensor, built on AlphaZero, was used to discover faster algorithms for matrix multiplication than the ones previously known [149] and optimize quantum circuit design [150].

For the basics of RL, we would recommend [43] or [151]. For a more practical approach, see [47] or [152]. We now first describe the formal RL framework in Section 2.2.1 and then explain different RL algorithms in Section 2.2.2.

2.2.1 Framework

The framework can be split into two parts: first, the description of the environment, detailing how the states change and which rewards are given under an action. This is done with the help of a Markov decision process (MDP). Second, the description of the agent, detailing which action it outputs in which state, which can be described using a *policy*.

The Environment

First, we define the environment. To do so, we need to define stochastic processes.

Definition 2.2 (Stochastic process [153]). *A* stochastic process *is a family $S = \{S_t\}_{t\in T}$ of random variables indexed by a parameter (usually the time) $t \in T$, where T is the* index set. *Its values are called* states, *and the set of possible states is called the* state space $\mathcal{S}$.

Examples of stochastic processes modeling physical systems are the weather, particle motion, or the number of cars parking in a parking lot at any given time [153].

In the following, we will take the parameter t to be the discretized time. For simplicity, we write down the definitions for finite action, reward, and state spaces. However, these definitions are easy to generalize by taking integrals instead of sums and "$\leq$" instead of "$=$" in probability distributions.

A special case of a stochastic process is a Markov process, where the next state depends on the current state and only the current state, not on the full history of states:

Definition 2.3 (Markov process (adjusted from [153] to fit [43])). *A* Markov process *is a stochastic process $S = \{S_t\}_{t\in T}$ with*

$$\mathbb{P}[S_{t+1} = s_{t+1}|S_t = s_t] = \mathbb{P}[S_{t+1} = s_{t+1}|S_1 = s_1, ..., S_t = s_t] \quad \forall\, t \in T$$

where $T \in \mathbb{N} \cup \{\infty\}$, $S_t \in \mathcal{S}$, and $\mathcal{S}$ is a finite set. The Markov process is called time-homogeneous *iff*

$$\mathbb{P}[S_{t+1} = s_{t+1}|S_t = s_t] = \mathbb{P}[S_1 = s_1|S_0 = s_0] \quad \forall\, t \in T.$$

A time-homogenous Markov process is then fully defined by the tuple $\langle \mathcal{S}, p\rangle$ and a starting state s_0 or a distribution q over them where

- *$\mathcal{S}$ is a finite set of states, and*
- *p is a transition probability defined via*

$$p(s'|s) = \mathbb{P}[S_{t+1} = s'|S_t = s] \quad \forall\, s, s' \in \mathcal{S}.$$

From now on, we will assume that our Markov processes are time-homogenous. With the following definition, we can now incorporate actions and rewards into our Markov processes.

Definition 2.4 (Markov decision process (MDP) (adjusted from [43] to fit [151, 153])). *A Markov decision process (MDP) is a time-homogeneous Markov process with associated stochastic variables $A_t \in \mathcal{A}$ and $R_t \in \mathcal{R}$ in each time step. Thereby, the* actions *A_t can influence the transition probabilities, and the* rewards *R_t have real values, i.e., $\mathcal{R} \subseteq \mathbb{R}$. A* discounted *MDP has a* discount factor *$\gamma \in [0, 1)$ associated with it. For an* undiscounted *MDP we set $\gamma = 1$. For an MDP, with an episode ending after $T \in \mathbb{N} \cup \{\infty\}$ timesteps, we define the* return *at time t as*

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k.$$

We call an MDP with $T \in \mathbb{N}$ episodic, *and one with $T = \infty$* continuing. *A finite MDP is fully defined by the tuple $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, p, \gamma\rangle$ where*

- $\mathcal{S}$ *is a finite set of states,*
- $\mathcal{R} \subseteq \mathbb{R}$ *is a finite set of rewards,*
- $\mathcal{A}$ *is a finite set of actions,*
- p *is a transition probability defined via*

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a].$$

  for all $s, s' \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}$*, and*
- $\gamma \in [0, 1]$ *is a discount factor.*

We included the discount factor so that the return can be properly defined for sequences that do not terminate, i.e., continuing tasks have to be discounted, and to be able to give immediate rewards more weight than later rewards. The return can be defined sequentially, i.e., we have

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k = R_{t+1} + \gamma \sum_{k=t+2}^{T} \gamma^{k-t-2} R_k = R_{t+1} + G_{t+1}.$$

MDPs are the usual formulation of environments in a mathematical framework. Often, instead of giving a transition probability as a probability over new states and rewards, we are given a state-transition probability

$$\tilde{p}(s'|s, a) := \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \tag{2.2.1}$$

and an expected reward for state-action pairs

$$r(s, a) := \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] \tag{2.2.2}$$

or state-action-state pairs

$$\tilde{r}(s, a, s') := \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s']. \tag{2.2.3}$$

Fully known finite MDPs can be mathematically *solved*, in the sense that we find an optimal policy, as we will discuss later. To do so, we only need to know the functions in Equation (2.2.1) and Equation (2.2.2) or (2.2.3). However, Definition 2.4 makes it clearer how to sample from an MDP, i.e., what happens in each agent-environment interaction. Furthermore, we can write

$$\tilde{p}(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a), \tag{2.2.4}$$

$$r(s, a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{2.2.5}$$

$$\tilde{r}(s, a, s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{\tilde{p}(s'|s, a)}, \tag{2.2.6}$$

i.e., we can write $\tilde{p}(s'|s, a)$, $r(s, a)$, $\tilde{r}(s, a, s')$ in terms of $p(s', r|s, a)$, which makes Definition 2.4 more general [43].

Sometimes, however, the underlying state is unknown to the agent, and the agent only has access to observations that are not a full Markov state. An environment like this can be described by a partially observable Markov decision process (POMDP), which is defined in the following [154].

Definition 2.5 (Partially observable Markov decision process (POMDP) (adjusted to fit [43] from [153,154]))**.** *A finite* partially observable Markov decision process (POMDP) *is a finite MDP with an additional stochastic observable variable, called* observation*, O_t assigned in every time step. It can be defined by a tuple $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, \Omega, p, \gamma \rangle$ where*

- *$\mathcal{S}$ is a finite set of states,*
- *$\mathcal{R} \subseteq \mathbb{R}$ is a finite set of rewards,*
- *$\mathcal{A}$ is a finite set of actions,*
- *Ω is a finite set of observations,*
- *p is a transition probability defined via*

$$p(s', r, o|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r, O_{t+1} = o|S_t = s, A_t = a]$$

  for all $s, s' \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}, o \in \Omega$, and

- *$\gamma \in [0, 1]$ is a discount factor.*

For POMDPs, we can introduce the notion of a *belief state*, which is a probability distribution over the states in $\mathcal{S}$ signifying the agent's belief on which state it is in. We can think of this as the state of a *belief MDP* [154].

We now have several ways on how an environment can be represented – depending on the information known to the agent, either as an MDP or a POMDP.

The Agent

Given an environment, the agent can decide what action to perform in the state it is in. We call the function it uses to make that decision its policy.

Definition 2.6 (Policy [43])**.** *Let $\mathcal{A}, \mathcal{S}$ be finite sets. A* policy *π is a map from a state $s \in \mathcal{S}$ to a probability distribution over the action space $\mathcal{A}$, denoted as $\pi(\cdot|s)$, i.e., we have*

$$\pi(\cdot|s) : \mathcal{A} \to [0, 1]$$

with

$$\sum_{a \in \mathcal{A}} \pi(a|s) = 1.$$

We write

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s].$$

We say π is deterministic *iff for all $s \in \mathcal{S}$ there exists $a_s \in \mathcal{A}$ such that $\pi(a|s) = \delta_{a,a_s}$. Then, we often write $\pi(s) = a_s$.*

Note that the state in this definition does not have to be a Markov state; it can also be an observation in a POMDP. The agent then wants to find the policy that maximizes the expected return in each state. To define the optimal policy, we need the notion of value functions [43].

Definition and Proposition 2.1 (Value functions [43, 155])**.** *Given a finite MDP with tuple* $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, p, \gamma \rangle$*, and a policy* π*, we define the* state-value function v_π *to be*

$$v_\pi(s) := \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s \right],$$

and the action-value function q_π *to be*

$$q_\pi(s,a) := \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s, A_t = a \right].$$

They fulfill the Bellman (expectation) equations

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \left(r + \gamma v_\pi(s') \right),$$

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \left(r + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \right).$$

We define the advantage function *to be*

$$\mathtt{A}_\pi(s,a) = q_\pi(s,a) - v_\pi(s).$$

Proof. The Bellman expectation equation for v follows from

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \left(r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \left(r + \gamma v_\pi(s') \right). \end{aligned}$$

The Bellman expectation equation for q follows similarly. □

By their definition, a value function tells us the expected return in a given state or of a given state-action pair. As such, they inform us how valuable a certain state or state-action pair is. Thus, it is no wonder that we can use them to define a notion of optimality.

Definition and Proposition 2.2 (Optimality [43])**.** *Let* $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, p, \gamma \rangle$ *define a finite MDP. Given two policies* π *and* π'*, we say* $\pi \geq \pi'$ *iff* $v_\pi(s) \geq v_{\pi'}(s)$ $\forall s \in \mathcal{S}$*. There always is at least one policy* π_* *s.t.* $\pi_* \geq \pi$ *for all other policies* π*. Such a policy is called an* optimal policy*. In case we do not want to distinguish them, we denote all optimal policies by* π_**; otherwise, we add an index. We define the* optimal state- *and* action-value functions *to be*

$$v_*(s) = \max_\pi v_\pi(s), \quad q_*(s,a) = \max_\pi q_\pi(s,a).$$

We then have

$$v_* = v_{\pi_*}, \quad q_* = q_{\pi_*}$$

for all optimal policies π^*. *The optimal value functions obey the* Bellman optimality equations

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v_*(s')\right),$$

$$q_*(s, a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma \max_{a' \in \mathcal{A}} q_*(s', a')\right).$$

This gives us a notion of optimality when considering different policies [43].

2.2.2 Algorithms

In RL, the agent has to find the optimal policy. The different algorithmic approaches to do so can be sorted into the following categories. The algorithms mentioned in this categorization as examples will be explained later in this section.

Model-based vs. model-free In very few cases, we know the MDP we want to solve. This can be seen as a model of the environment. Then, we can use the information provided by the model to find the optimal policy, e.g., with dynamic programming. Sometimes, we are provided with an approximate model of the environment, and sometimes, an algorithm solves the RL task by building/learning a model and then optimizing the policy in that given model. Algorithms like this are called *model-based*. Other algorithms, known as *model-free* algorithms, do not make use of any model and are trial-and-error learners interacting directly with the environment. Most algorithms we will discuss here are model-free algorithms. We can also use hybrid algorithms, i.e., combinations of model-free and model-based algorithms. Examples include Dyna-Q and I2A, which we will not discuss in this section [43, 47].

Value-based vs. policy-based Some algorithms, known as *value-based* algorithms, first learn the value function and generate policies based on that. This is usually done iteratively. They are commonly built on dynamic programming (for small state and action spaces) and SARSA. Most algorithms today use NNs to approximate the Q-function, in the form of deep Q-networks (DQNs), as a basis. Other algorithms directly optimize the policy w.r.t. the return. They are called *policy-based*. REINFORCE forms the foundation for most of these algorithms. Both have their advantages and disadvantages: Policy-based algorithms are more general and hence easier to use in infinite action spaces (at least in their standard definition, the above-mentioned value-based algorithms cannot be used for infinite action spaces), and they are guaranteed to converge to a local optimum. On the other hand, value-based algorithms are, generally speaking, more sample-efficient and have a lower variance. Again, there are a lot of hybrid models that make use of both of these methods, e.g., actor-critic methods like A2C, PPO, and SAC [47].

On-policy vs. off-policy Some RL algorithms are trained on-policy, i.e., the current policy can only be updated with samples generated from that same policy. Other algorithms, however, can also be trained off-policy. That is, the current policy can also be updated with samples generated from other policies. Off-policy algorithms are generally much more sample-efficient as training samples can be used several times instead of only once [47].

First, we will explore dynamic programming, which can be used for small-size, known MDPs. Secondly, we will look at the algorithms developing from that, i.e., SARSA and DQNs, which can be used for unknown environments, either again only for small-size MDPs or small action spaces, respectively. Thirdly, we are looking at REINFORCE as the basis of policy-based algorithms, and lastly, we will explore combined algorithms like PPO, TD3, and SAC.

Dynamic Programming

For this section, assume that we are given an MDP with the tuple $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, p, \gamma \rangle$, and $|\mathcal{S}|, |\mathcal{A}|$ are relatively small. In this case, we can always find a deterministic optimal policy. The algorithm for finding this optimal policy is split into two parts: evaluating and improving the current policy. We use [43] for this part.

Evaluating the policy To evaluate a policy π, we make use of the Bellman expectation equation for the state-value function

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v_\pi(s')\right).$$

This is a system of $|\mathcal{S}|$ linear equations of $|\mathcal{S}|$ variables, which are the $v_\pi(s)$. In our case, it is best to use an iterative solution method. Let $v_0, v_1, v_2, \ldots$ be a sequence of approximations for v_π. We can choose v_0 as we like, except for the terminal state. If there is one, it has to have the value 0, as the episode ends after reaching the terminal state, and hence no reward can be obtained after reaching it. Then we obtain the approximation v_{k+1} from v_k by setting

$$v_{k+1} = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v_k(s')\right).$$

The point $v_k = v_\pi$ is a fixed point in this search, and we can show that the sequence $\{v_k\}$ converges to v_π for $k \to \infty$. We call this algorithm *iterative policy evaluation*. As one can see, the updates are performed w.r.t. the expectation value. That is why they are called *expected* updates. In practice, we usually update the v_k in-place, i.e., if we already updated the values for some of the states, we use these updated versions to update the values for other states. This makes the algorithm converge faster. However, the convergence rate heavily depends on which states are evaluated first. Also, we usually only update until $\max_{s \in \mathcal{S}} |v_k(s) - v_{k+1}(s)| < \theta$ for a small threshold $\theta > 0$, so that the algorithm, combined with improving the policy, converges faster.

Improving the policy How we improve our policy is largely based on the following theorem.

Theorem 2.2 (Policy improvement Theorem [43])**.** *Let $\langle \mathcal{S}, \mathcal{R}, \mathcal{A}, p, \gamma \rangle$ define a finite MDP. Given two deterministic policies π and π' with $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ for all $s \in \mathcal{S}$, we know $\pi' \geq \pi$.*

Hence, the policy π' we choose after evaluating v_π for our current policy π is defined by

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v_\pi(s')\right)$$

where we abuse the notation $\arg\max_{a \in \mathcal{A}}$ slightly and choose one of the actions maximizing $q_\pi(s, a)$. This kind of policy update is called *greedy* as we always choose the best action

possible for the state we are in, according to the given value function. If we have $v_\pi = v_{\pi'}$, it is

$$v_{\pi'}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v_{\pi'}(s')\right)$$

which is the Bellman optimality equation, i.e. π and π' are optimal and we can stop our iterative process. Although we only considered deterministic policies, the results easily carry over to non-deterministic policies.

Policy iteration We then iterate over both algorithms. After initializing a policy, we always first evaluate and then improve the policy until we reach a steady state. This is detailed in Algorithm 1.

Algorithm 1 Policy Iteration for finding an optimal policy [43]

Choose $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}$ arbitrarily $\forall s \in \mathcal{S}$ ▷ *Initialization*
Set $v(s_{\text{terminal}}) \leftarrow 0$ for any terminal state s_{terminal}
$policy - stable \leftarrow False$
$\Delta \leftarrow \theta$
while not $policy - stable$ **do** ▷ *Policy Iteration*
  while $\Delta \geq \theta$ **do** ▷ *Policy Evaluation*
    $\Delta \leftarrow 0$
    for $s \in \mathcal{S}$ **do**
      $v \leftarrow v(s)$
      $v(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v(s')\right)$
      $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
  $policy - stable \leftarrow False$
  for $s \in \mathcal{S}$ **do** ▷ *Policy Improvement*
    $a_{\text{old}} \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left(r + \gamma v(s')\right)$
    If $\pi(s) = a_{\text{old}}$ then $policy - stable \leftarrow True$
return $v \approx v_*$ and $\pi \approx \pi_*$

If the MDP is not known anymore, we can, e.g., use SARSA.

SARSA

Now, let us assume that we still have a small-scale MDP, but we do not know the transition probabilities anymore. Instead, we interact with the environment, i.e., sample from it. This section is adapted from [43, 151].

Evaluating the policy In this case, we will directly evaluate the action-value function to make it easier. Of course, we cannot use the Bellmann expectation equations anymore to do so as we cannot calculate the expectation values. Instead, we can use the actual returns or rewards. *Monte-Carlo (MC)* methods wait for the end of the episode and then update the value q of each visited state S_t and action A_t via

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(G_t - q(S_t, A_t))$$

where α is a hyperparameter. If we choose α to be one over the number of times the state has been visited, we get $q(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a]$ in the limit of infinite runs,

which coincides with the definition of q in Definition and Proposition 2.1. Another possible protocol is to update the value function in each time step using the estimate of the return $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$ instead of the observed return. We then get the update

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)).$$

This kind of update is called a *temporal difference (TD(0))* update, and we call

$$\delta_t = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)$$

the *TD-error*. The TD approach has the upside of not having to wait for the end of the episode to update the value function, which makes it more efficient. It also has a lower variance than the MC approach, but it has a higher bias to the initial value function. We can get an in-between-picture when we consider the *n-step return*

$$G_{t:t+n} = \sum_{k=t+1}^{t+n} \gamma^{k-t-1} R_k + \gamma^n q(S_{t+n}, A_{t+n}),$$

and update our value function w.r.t. this, i.e.,

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(G_{t:t+n} - q(S_t, A_t)).$$

Using these n-step returns, we define the λ-*return*

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

for $\lambda \in [0, 1]$. This gives rise to the update

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha(G_t^\lambda - q(S_t, A_t))$$

and TD(λ). Instead of this forward view of TD(λ) and having to wait until a number of rewards happen before we can update our value function, we can take a backward approach: We keep track of all states we visited. In each step, using the reward in that step, we update the value for all visited states and actions that could have influenced the reward. We achieve this with the help of *eligibility traces* defined by

$$\begin{aligned} E_0(s, a) &= 0, \\ E_t(s, a) &= \gamma\lambda E_{t-1}(s, a) + \delta_{S_t, s}\delta_{A_t, a}. \end{aligned}$$

In each timestep, we then perform the updates

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t), \\ E_t(s, a) &= \gamma\lambda E_{t-1}(s, a) + \delta_{S_t, s}\delta_{A_t, a}, \\ q(S_t, A_t) &\leftarrow q(S_t, A_t) + \alpha\delta_t E_t(S_t, A_t). \end{aligned}$$

This is then called TD(λ) in its backward view.

Improving the policy In improving the policy, now that we are not given the full MDP anymore and can only sample from our environment, we are faced with the *exploration vs. exploitation* trade-off. This means that in each state, we have to decide between exploiting

what we already know, i.e., taking the action with the best mean return until now, and exploring different options, i.e., taking an action we have not performed as often until now. There are a lot of different examples of policies aiming at balancing this trade-off. One possible policy that also explores but for which a kind of policy improvement theorem still holds is the ϵ-*greedy* policy for $\epsilon \in (0, 1)$. We still act greedily with probability $1 - \epsilon$ but explore and choose a random action with probability ϵ, i.e., we choose

$$\pi(s|a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a = a_* = \arg\max_{a \in \mathcal{A}} q(s, a) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{else} \end{cases}$$

as our policy.

Policy iteration We then again iterate over both the policy evaluation and improvement, but now, we usually do the iteration after every timestep to ensure faster convergence. This is called *SARSA(λ)* and shown in Algorithm 2.

Algorithm 2 SARSA(λ) [43, 151]

Choose $q(s, a) \in \mathbb{R}$ arbitrarily $\forall s \in \mathcal{S}, a \in \mathcal{A}$ ▷ *Initialization*
Set $q(s_{\text{terminal}}, \cdot) \leftarrow 0$ for any terminal state s_{terminal}
while True **do**
  $E(s, a) \leftarrow 0 \ \forall s \in \mathcal{S}, a \in \mathcal{A}$
  Choose $S \in \mathcal{S}$ at random from the starting state distribution
  Choose $A \in \mathcal{A} \sim \pi(\cdot|S)$, where π is derived from q, e.g. ϵ-greedy
  while $S \neq S^{\text{terminal}}$ **do**
    Take action A, observe reward R and state S'
    Choose $A' \in \mathcal{A} \sim \pi(\cdot|S')$, where π is derived from q, e.g. ϵ-greedy
    $\delta \leftarrow R + \gamma q(S', A') - q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$
    for $s \in \mathcal{S}, a \in \mathcal{A}$ **do**
      $q(s, a) \leftarrow q(s, a) + \alpha \delta E(s, a)$
      $E(s, a) \leftarrow \gamma \lambda E(s, a)$
    $S, A \leftarrow S', A'$

If the state space is not finite anymore, we need to use different algorithms such as DQNs.

Deep Q-Network (DQN)

Now let us assume we are given an unknown MDP with a small-size action space $\mathcal{A}$ but a large, possibly infinite, state space $\mathcal{S}$. Then, we can not use SARSA(λ) in its tabular form anymore. Instead, we can use function approximation to approximate q. In principle, we can use any function approximator to do this, although the most common ones are linear approximators or NNs [151]. Here, we want to talk about approximating the q function with NNs. Its parameters, i.e., weights and biases, we will call θ. Also, instead of $R + \gamma q(S', A') - q(S, A)$, we will use $R + \gamma \max_{a \in \mathcal{A}} q(S', a) - q(S, A)$ as the TD-error. This makes it possible to learn off-policy about the greedy policy while using, e.g., an ϵ-greedy policy in practice. For training the NN, we use a series of quadratic loss functions

$$c_i(q_\theta, ((s_i, a_i), y_i) = (y_i - q_\theta(s_i, a_i))^2 .$$

Here and from now on, s does not have to be a Markov state but can also only be features of the state or observations [118].

Experience replay As this algorithm can be used off-policy, we are not restricted to using each sample once, but we can use them again in a later round of training. Each state-action-state-reward tuple is saved in a *replay buffer*, which is in turn used to optimize the DQN. This process is called *experience replay* and leads to Algorithm 3 [118]. Today, there are different, more sophisticated variants of experience replay, e.g., prioritized experience replay (PER) [156] and hindsight experience replay (HER) [157].

Algorithm 3 DQN with Experience Replay [118]

$\mathcal{D} \leftarrow \{\}$ with capacity N ▷ *Initialization*
Choose weights θ for q_θ at random
while True **do**
  Choose $S \in \mathcal{S}$ at random from the starting state distribution
  Choose $A \in \mathcal{A} \sim \pi(\cdot|S)$, where π is derived from q_θ, e.g. ϵ-greedy
  while $S \neq S^{\text{terminal}}$ **do**
    Take action A, observe reward R and state S'
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(S, A, R, S')\}$
    Sample random minibatch $\{(S_i, A_i, R_i, S_i')\}_{i=1,\dots,m}$ from $\mathcal{D}$
    Update θ according to the stochastic gradient descent step

$$\theta \leftarrow \theta + \frac{\eta}{m} \sum_{i=1}^{m} \left(R_i + \gamma \max_{a \in \mathcal{A}} q_\theta(S_i', a) - q_\theta(S_i, A_i) \right) \nabla_\theta q_\theta(S_i, A_i)$$

    $S \leftarrow S'$
    Choose $A \in \mathcal{A} \sim \pi(\cdot|S)$, where π is derived from q, e.g. ϵ-greedy

Double DQN In the vanilla DQN algorithm, we use the same value function approximator to select actions and evaluate these actions. This can lead to overestimating their values. Instead, we can use different weights, θ and θ', for those two tasks and use

$$y_i = R_i + \gamma q_{\theta'} \left(S_i', \arg \max_{a \in \mathcal{A}} q_\theta \left(S_i', a \right) \right)$$

as a target for input (S_i, A_i). For example, we can update the target network less frequently than the one giving rise to the policy but still use the same network. This leads to better convergence [158]. Other algorithms like dueling DQNs are built on this [159].

REINFORCE

Now consider the general case of an unknown MDP with arbitrary $\mathcal{S}, \mathcal{A}$. REINFORCE is the classic policy gradient algorithm first presented in [160]. We directly parametrize the policy π_θ, e.g., using a NN, with parameters θ. We then use this policy to generate a trajectory $\tau = S_0, A_0, R_1, S_1, ..., A_{T-1}, R_T, S_T$ where S_T is the terminal state or T is the episode length. We want to maximize the expectation value of the return G_0, i.e., we want to find the parameters θ that maximize

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G_0(\tau)].$$

This function to maximize is called the *objective function*. Its gradient is given by

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right].$$

This is the same gradient we get when considering the often-used objective function

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G_t \log \pi_\theta(A_t|S_t)\right].$$

Again, because the MDP is unknown, we cannot calculate the above expectation value but have to sample from the MDP. Updating the policy after every episode leaves us with the Algorithm 4. From the definition, it is clear that this algorithm is an on-policy algorithm that cannot use past data for updates [47].

Algorithm 4 REINFORCE [47]

Choose NN architecture and learning rate η ▷ *Initialization*
Choose weights θ for π_θ at random π_θ
while True **do**
  Sample trajectory $\tau \leftarrow S_0, A_0, R_1, S_1, ..., A_{T-1}, R_T, S_T$ according to π_θ
  $G_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t-1} R_k$ for $t = 0, ..., T-1$
  $\nabla_\theta J(\theta) \leftarrow \sum_{t=0}^{T} G_t \nabla_\theta \log \pi_\theta(A_t|S_t)$
  $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$

This algorithm uses an unbiased estimate for the policy gradient, which leads to high variance. To reduce this variance (but introduce a bit of bias), we often subtract a baseline b in each state and use

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} (G_t - b(S_t)) \log \pi_\theta(A_t|S_t)\right]$$

as our objective function. One example of this is using the value function as a baseline, which is, e.g., one of the tricks in advantage actor-critic (A2C) [47].

Another way to improve the performance and help with exploration is to add an entropy term

$$h_\theta(a|s) = -\ln \pi_\theta(a|s)$$

to the objective function as first introduced in [161]. The objective function then becomes

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} (G_t - b(S_t)) \log \pi_\theta(A_t|S_t) + \beta h_\theta(A_t|S_t)\right]$$

where β is a hyperparameter. As the entropy is higher the more uniform the policy is, this reinforces exploration. The hyperparameter β can help to balance exploration with exploitation. This is also called *entropy regularization*, also, e.g., used in A2C, which we will discuss now [47, 161].

Advantage Actor-Critic (A2C)

Actor-critic algorithms combine the ideas of REINFORCE with the ones of (double) DQNs. We have two NNs, the *actor* representing the policy π_{θ_A} with parameters θ_A and the *critic* representing the state-value function v_{θ_C} with parameters θ_C. This helps to stabilize policy gradient algorithms while still keeping a lot of their advantages. Advantage actor-critic (A2C) algorithms use the advantage as the parameter to optimize. Given an estimate for the value function $v_{\theta_C} \approx v_{\pi_{\theta_A}}$, we get approximately

$$\mathtt{A}_{\pi_{\theta_A}}(s_t, a_t) \approx G_t - v_{\theta_C}(s_t)$$

with $n < T$ for the advantage $\mathtt{A}_{\pi_{\theta_A}}$. Our objective function for the actor NN then becomes

$$J(\theta_A) = \mathbb{E}_{\tau \sim \pi_{\theta_A}} \left[\sum_{t=0}^{T} (G_t - v_{\theta_C}(S_t)) \log \pi_{\theta_A}(A_t|S_t) + \beta h_{\theta_A}(A_t|S_t) \right]$$

and our cost for training the critic after each episode is given by

$$C(\theta_C) = \sum_{t=0}^{T} (G_t - v_{\theta_C}(S_t))^2 .$$

The algorithm following from this is presented in Algorithm 5. As a generalization, in the case where our environment is replicable, we can use asynchronous advantage actor-critic (A3C). There, we let each episode run on several copies of the same environment to better estimate the gradients for the updates [47, 162].

Algorithm 5 Advantage actor-critic (A2C) [47, 162]

Choose weights θ_C for v_{θ_C} and θ_A for π_{θ_A} at random ▷ *Initialization*
while True **do**
  $t \leftarrow 0$
  Choose $S_0 \in \mathcal{S}$ at random from the starting state distribution
  while $S_t \neq S^{\text{terminal}}$ or $t < T$ **do**
    $t \leftarrow t + 1$
    Choose $A_t \in \mathcal{A} \sim \pi_{\theta_A}(\cdot|S)$
    Take action A_t, observe reward R_{t+1} and state S_{t+1}
  $y \leftarrow 0$
  $y \leftarrow v_{\theta_C}(S_T)$ if $S_T \neq S^{\text{terminal}}$
  for i=t-1,...,0 **do**
    $y \leftarrow \gamma y + R_i$
    $d\theta_C \leftarrow d\theta_C + \nabla_{\theta_C}(y - v_{\theta_C}(S_i))^2$
    $d\theta_A \leftarrow d\theta_A + \nabla_{\theta_A}(\log \pi_{\theta_A}(A_i|S_i)(y - v_{\theta_C}(S_i)) + \beta h_\theta(A_t|S_t))$
  Update θ_C, θ_A using gradients $d\theta_C, d\theta_A$

Trust Region Policy Optimization (TRPO)

Both presented algorithms using policy gradients have a major problem: Sometimes, the performance collapses because a small change in parameters can lead to a big change in the policy. This unsuccessful trajectory is then used to generate more data, which can lead to an even worse policy, which can make it harder to recover from the performance collapse.

Trust region policy optimization (TRPO) is a way of dealing with this problem. Instead of directly optimizing the return, we optimize the change in return

$$J(\pi') - J(\pi) \approx \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \mathtt{A}^{\pi}(S_t, A_t)\frac{\pi'(A_t|S_t)}{\pi(A_t|S_t)}\right] =: J_{\pi}^{\mathrm{CPI}}(\pi')$$

where π is our old policy, π' our new one and CPI stands for conservative policy iteration. This is called a *surrogate objective*. The error in this approximation is bounded by the expectation value of the KL divergence, which is given by

$$\mathrm{KL}(\pi'(a|s)\|\pi(a|s)) = \sum_{a\in Acal} \pi'(a|s)\frac{\pi'(a|s)}{\pi(a|s)}$$

for finite $\mathcal{A}$ [109]. So, if we do not want our policy to get much worse in one training step, the KL divergence has to be small enough. This leads to an algorithm called TRPO, which was first presented in [155], where we consider the optimization problem

$$\begin{aligned}&\max_{\theta} \mathbb{E}_t\left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\mathrm{old}}}(A_t|S_t)}\mathtt{A}_t^{\pi_{\theta_{\mathrm{old}}}}\right]\\ &\mathbb{E}_t\left[\mathrm{KL}(\pi_\theta(a|s)\|\pi_{\theta_{\mathrm{old}}}(a|s))\right] \leq \delta,\end{aligned}$$

i.e., we maximize the surrogate objective while keeping the KL divergence small. We can use this objective both for REINFORCE and Actor-Critic algorithms [47].

Proximal Policy Optimization (PPO)

Proximal policy optimization (PPO) aims at keeping the data efficiency and reliable performance of TRPO while simplifying its implementation. We want to incorporate the constraint on the KL divergence directly in the objective function. This is done by using the objective

$$J(\theta) = \mathbb{E}_t\left[\min\left(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\mathrm{old}}}(A_t|S_t)}\mathtt{A}_t^{\pi_{\theta_{\mathrm{old}}}}, \mathrm{clip}\left(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\mathrm{old}}}(A_t|S_t)}\mathtt{A}_t^{\pi_{\theta_{\mathrm{old}}}}, 1-\epsilon, 1+\epsilon\right)\right)\right]$$

in actor-critic or other policy-gradient-based algorithms. Hereby, ϵ is a hyperparameter and

$$\mathrm{clip}\left(x, 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1-\epsilon & \text{if } x < 1-\epsilon \\ x & \text{if } 1-\epsilon \leq x \leq 1+\epsilon \\ 1+\epsilon & \text{if } x > 1+\epsilon \end{cases}.$$

This clipping function makes sure that the policy does not change too much in each timestep, just like the constraint in TRPO. We will not write down the algorithm as its form basically stays the same as in Algorithm 5. Only the objective gets replaced [163].

Deep Deterministic Policy Gradient(DDPG)

Until now, the presented algorithms available for continuous actions are on-policy algorithms and do not use a replay buffer, making them sample-inefficient. The only ones that did were the purely value-based DQN variants. An algorithm changing that is deep deterministic policy gradient (DDPG). As in the DQN, we use a NN with parameter θ_C to approximate the value function and use experience replay to train it. However, instead of acting ϵ-greedily w.r.t. q_{θ_C}, which is not easily possible for infinite action spaces, we also parameterize the

deterministic policy π_{θ_A} and add a little continuous noise for exploration. We use a policy gradient to update π_{θ_A}, which is purely based on q_{θ_C} and no real-world returns. This makes it possible to perform the algorithm off-policy and thus makes it more sample-efficient than the policy gradient-based algorithms considered before [164].

We will not present the algorithm here as it is very similar to TD3, which is presented next. In Algorithm 6, to get DDPG, just replace the two critics with one critic, ignore the target networks, and set $d = 1$.

Twin Delayed Deep Deterministic Policy Gradient (TD3)

As for the DQN, we also get a problem with overestimating values in DDPG. That is why the following changes were made to DDPG, in order to get twin delayed deep deterministic policy gradient (TD3) as presented in Algorithm 6.

Algorithm 6 TD3 [103]

$\mathcal{D} \leftarrow \{\}$ with capacity N ▷ *Initialization*
Choose weights $\theta_{C1}, \theta_{C2}, \theta_A$ for $q_{\theta_{C1}}, q_{\theta_{C2}}, \pi_{\theta_A}$ at random
$\theta'_{C1} \leftarrow \theta_{C1}, \theta'_{C2} \leftarrow \theta_{C2}, \theta'_A \leftarrow \theta_A$ ▷ *Target Networks*
$n = 0$ ▷ *Step Counter*
while True **do**
  Choose $S \in \mathcal{S}$ at random from the starting state distribution
  $A \leftarrow \pi_{\theta_A}(S) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma)$ and $\mathcal{N}$ is the exploration noise
  while $S \neq S^{\text{terminal}}$ **do**
    $n \leftarrow n + 1$
    Take action A, observe reward R and state S'
    Choose $A' \leftarrow \pi_{\theta_A}(S) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(S, A, R, S')\}$
    $S, A \leftarrow S', A'$
    Sample random minibatch $\{(S_i, A_i, R_i, S'_i)\}_{i=1,\dots,m}$ from $\mathcal{D}$
    for i=1,...,m **do**
      $\tilde{A}_i \leftarrow \pi_{\theta'_A}(S) + \epsilon$ where $\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c)$
      $y_i \leftarrow R_i + \gamma \min_{j=1,2} q_{\theta'_{Cj}}(S'_i, \tilde{A}_i)$
    Update θ_{Cj} according to loss $L_{Cj} = \frac{1}{m} \sum_{i=1}^{m} \left(y_i - q_{\theta_{Cj}}(S_i, A_i)\right)^2$ ▷ *Update critics*
    if n mod d **then** ▷ *Update targets and actor only every d steps*
      $\theta_A \leftarrow \theta_A + \eta \frac{1}{m} \nabla_{a_i} q_{\theta_{C1}}(s_i, a_i)|_{a=\pi_{\theta_A}(s_i)} \nabla_{\theta_A} \pi_{\theta_A}(s_i)$ ▷ *Update actor*
      $\theta'_{Cj} \leftarrow \tau\theta_{Cj} + (1 - \tau)\theta'_{Cj}$ ▷ *Update target critics*
      $\theta'_A \leftarrow \tau\theta_A + (1 - \tau)\theta'_A$ ▷ *Update target actor*

Firstly, the policy is updated only every d steps, i.e., less often than the value function. Secondly, a second critic is introduced so that, when updating the value function, we always take the minimum over both critics as a target, and do not overestimate the Q-value, similarly to double DQN. Thirdly, the target is further modified by introducing target parameters that are also only updated every d steps smoothly, i.e., we do not replace the target parameters with the current parameters but incrementally adjust the target parameters towards the current parameters. Lastly, we not only use the target parameters and both critics to update the value network, but we also do it w.r.t. a clipped target action sampled from the

policy plus noise that is clipped to certain boundaries. This improves the performance in a lot of tasks by reducing overestimation bias [103].

Soft Actor-Critic (SAC)

Soft actor-critic (SAC) was published around the same time as TD3 and aims at extending DDPG to stochastic policies while reducing its overestimation bias. It is based on the maximum entropy framework, where we try to maximize

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}\left[R_t + \beta h(A_t|S_t)\right].$$

In this kind of setting, we can look at the *soft* value function, recursively defined via

$$Q(s,a) = \mathbb{E}_{r,s'\sim p(\cdot,\cdot|s,a)}\left[r + \gamma V(s')\right]$$

and

$$V(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\left[Q(s,a) - \log\pi(a|s)\right],$$

instead of the value function. Note that this differs from our earlier definition of the value function by the inclusion of the term $\log\pi(a|s)$. In the original paper [100], the soft state-value function V_{θ_v}, a target soft state-value function $V_{\theta_v'}$, the soft action-value function Q_{θ_q} and the policy π_{θ_π} are parametrized. All seen state-action-reward-state tuples are again stored in a replay buffer $\mathcal{D}$. As objective functions, we use quadratic losses for the critics, i.e.,

$$\begin{aligned} J_v(\theta_v) &= \frac{1}{2}\mathbb{E}_{s\sim\mathcal{D}}\left[\left(V_{\theta_v} - \mathbb{E}_{a\sim\pi_{\theta_\pi}(\cdot|s)}\left[Q_{\theta_q}(s,a) - \log\pi_{\theta_\pi}(a|s)\right]\right)^2\right] \\ J_q(\theta_q) &= \frac{1}{2}\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q_{\theta_q}(s,a) - r - \gamma V_{\theta_v'}(s')\right)^2\right]. \end{aligned}$$

For the actor, we employ the KL divergence between the given policy and the *softmax* [37,47] policy

$$\pi_{\text{softmax}}(\cdot|s) = \frac{\exp Q_{\theta_q}(s,\cdot)}{Z_{\theta_q}(s)}, \quad Z_{\theta_q}(s) = \sum_a \exp Q_{\theta_q}(s,a)$$

of the soft Q-values, i.e.,

$$J_\pi(\theta_\pi) = \mathbb{E}_{s\sim\mathcal{D}}\left[\text{KL}\left(\pi_{\theta_\pi}(\cdot|s)\|\frac{\exp Q_{\theta_q}(s,\cdot)}{Z_{\theta_q}(s)}\right)\right].$$

Together, we get Algorithm 7, where we calculated some gradients more explicitly [100].

In newer versions of the algorithm, only the action-value function is parametrized, and similar tricks as in TD3, i.e., using two parametrizations of Q functions, separate target Q functions, and minimizing over the two target networks to get the target, are employed [101].

Truncated Quantile Critics (TQC)

A way to deal with the overestimation of the Q function we discussed earlier is to use two critic networks and use their minimum as a target for updating them. Truncated quantile critics (TQC), as presented in [102], adds another layer to this: for the actor, we still

Algorithm 7 SAC [100]

$\mathcal{D} \leftarrow \{\}$ with capacity N
Choose weights $\theta_v, \theta_q, \theta_\pi$ for $V_{\theta_v}, Q_{\theta_q}, \pi_{\theta_\pi}$ at random
$\theta'_v \leftarrow \theta_v$
while True **do**
  Choose $S \in \mathcal{S}$ at random from the starting state distribution
  Choose $A \in \mathcal{A} \sim \pi_{\theta_\pi}(\cdot|S)$
  for each environment step **do**
    Take action A, observe reward R and state S'
    Choose $A \in \mathcal{A} \sim \pi_{\theta_\pi}(\cdot|S)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(S, A, R, S')\}$
    $S, A \leftarrow S', A'$
  for each gradient step **do**
    Sample random minibatch $\tilde{\mathcal{D}}$ from $\mathcal{D}$
    $\theta_v \leftarrow \theta_v - \eta_v \mathbb{E}_{s\sim\tilde{\mathcal{D}}}\left[\left(\nabla_{\theta_v} V_{\theta_v}(s)\right)\left(V_{\theta_v}(s) - \mathbb{E}_{a\sim\pi_{\theta_\pi}(s,\cdot)}\left[Q_{\theta_q}(s,a) - \log \pi_{\theta_\pi}(a|s)\right]\right)\right]$
    $\theta_q \leftarrow \theta_q - \eta_q \mathbb{E}_{(s,a,r,s)\sim\tilde{\mathcal{D}}}\left[\left(\nabla_{\theta_q} Q_{\theta_q}(s,a)\right)\left(Q_{\theta_q}(s,a) - r - \gamma V_{\theta'_v}(s')\right)\right]$
    $\theta_\pi \leftarrow \theta_\pi - \eta_\pi \mathbb{E}_{(s,a)\sim\tilde{\mathcal{D}}}\left[\nabla_{\theta_\pi} \mathrm{KL}\left(\pi_{\theta_\pi}(\cdot|s) \| \frac{\exp Q_{\theta_q}(s,\cdot)}{Z_{\theta_q}(s)}\right)\right]$
    $\theta'_v \leftarrow \tau\theta_v + (1-\tau)\theta'_v$

parametrize the policy π as usual with a parameter ϕ. For the critic, instead of the value function, we estimate the distribution over return values $G^\pi(s,a)$. In particular, we train N networks $\theta_{\psi_n} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^M$, $n = 1, ..., N$ to estimate $G^\pi(s,a)$ with a mixture of M *atoms* (first introduced in [165] for DQNs)

$$G_{\psi_n}(s,a) = \frac{1}{M}\sum_{m=1}^{M} \delta(\theta^m_{\psi_n}(s,a)),$$

i.e., Dirac delta distributions at positions $(\theta^m_{\psi_n}(s,a))_{m=1,...,M}$. This is called a *quantile* distribution. To get the target distribution, we pool these atoms into one set

$$\mathcal{G}(s',a') := \{\theta^m_{\psi'_n}(s',a') | n = 1, ..., N,\ m = 1, ..., M\}$$

for target networks $(\theta_{\psi'_n})_{n=1,...,N}$. Sorting them in ascending order, we name $\mathcal{G}$'s elements $g_i(s,a)$ for $i = 1, ..., MN$. We then *truncate* the set by choosing the kN smallest elements (where $k \in \{1, ..., M\}$) of $\mathcal{G}(s',a')$ to define atoms

$$y_i(s,a) := r + \gamma\left(z_i(s',a') - \beta \log \pi_\phi(a'|s')\right)$$

of the target distribution

$$Y(s,a) = \frac{1}{kN}\sum_{i=1}^{kN} \delta(y_i(s,a)).$$

We then minimize the 1-Wasserstein distance between $G_{\psi_n}(s,a)$ and $Y(s,a)$ by *quantile regression*. Using the Huber quantile loss leads to the objective function

$$J_G(\psi_n) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\frac{1}{kNM}\sum_{m=1}^{M}\sum_{i=1}^{kN} \rho^H_{\tau_m}(y_i(s,a) - \theta_{\psi_n}(s,a))\right]$$

where

$$\rho_\tau^H(u) = |\tau - \mathbb{1}_{u<0}| \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq 1 \\ (|u| - \frac{1}{2}), & \text{else} \end{cases}, \quad \tau_m = \frac{2m-1}{2M}, \quad m = 1, ..., M.$$

For updating the policy, we use the full ensemble $\mathcal{G}(s, a)$ and the objective

$$J_\pi(\phi) = \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_\phi(\cdot|s)} \left[\beta \log \pi_\phi(a|s) - \frac{1}{NM} \sum_{m,n=1}^{M,N} \theta_{\psi_n}^m(s, a) \right].$$

We dynamically update the entropy temperature coefficient β using gradient steps with the objective

$$J(\beta) = -\mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_\phi(\cdot|s)} \left[\log \beta \left(\log \pi_\phi(a|s) + \mathcal{H}_T \right) \right]$$

where the target entropy $\mathcal{H}_T$ is heuristically set to $\dim \mathcal{A}$ [102, 165]. The full algorithm is presented in Algorithm 8.

Algorithm 8 TQC [102]

$\mathcal{D} \leftarrow \{\}, \mathcal{H}_T \leftarrow -\dim \mathcal{A}, \beta \leftarrow 1, \alpha \leftarrow 0.005$
Choose weights ϕ, ψ_n, ψ'_n for $\pi_\phi, G_{\psi_n}, G_{\psi'_n}$, $n = 1, ..., N$ at random
for each iteration **do**
  Choose $S \in \mathcal{S}$ at random from the starting state distribution
  Choose $A \in \mathcal{A} \sim \pi_{\theta_\pi}(\cdot|S)$
  for each environment step **do**
    Take action A, observe reward R and state S'
    Choose $A \in \mathcal{A} \sim \pi_{\theta_\pi}(\cdot|S)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(S, A, R, S')\}$
    $S, A \leftarrow S', A'$
  for each gradient step **do**
    Sample random minibatch $\tilde{\mathcal{D}}$ from $\mathcal{D}$
    $\beta \leftarrow \beta - \eta_\beta \nabla_\beta J(\beta)$
    $\phi \leftarrow \phi - \eta_\pi \nabla_\phi J_\pi(\phi)$
    $\psi_n \leftarrow \psi_n - \eta_G \nabla_{\psi_n} J_G(\psi_n),\ n = 1, ..., N$
    $\psi'_n \leftarrow \alpha\psi_n + (1 - \alpha)\psi'_n,\ n = 1, ..., N$
return $\pi_\phi, G_{\psi_n}, n = 1, ..., N$

3

Quantum Information

Solving some problems on a classical computer needs more than polynomial resources in problem size. Examples of problems for which no effective classical algorithm guarantees a solution but whose solutions can be quickly checked are factorizing an integer into prime numbers or the traveling salesman problem. Furthermore, simulating general quantum systems is hard for classical computers. With the aim of solving the last problem, in 1982, R. Feynman proposed the idea of using a fixed quantum system to simulate any quantum system and called this a *quantum computer (QC)* [52]. Where classical computers use bits and electrical circuits in computations, quantum computers rely on quantum mechanical systems, e.g., qubits, to perform calculations. The field of quantum computing (QC) is a sub-field of *quantum information (QI)*, which arose in the 1970s and 1980s in discussions on the use of quantum mechanical systems as an informational resource instead of only a way to explain natural phenomena [51].

Starting in the 1990s, quantum algorithms which, at least theoretically, outperform the best-known classical algorithms, e.g., the Deutsch-Josza algorithm [53] for determining if a function on bits is constant or balanced (i.e., 1 for half of the inputs and 0 for the other half), Grover's algorithm for unstructured searches [54], or Shor's algorithm [55] for factorizing an integer into prime numbers, were developed. In the last years, significant progress has been made on the experimental realization of quantum computers [58–63].

Most sections of this chapter (Sections 3.1-3.5, 3.7, 3.10) are aimed at readers who are not familiar with QI and establish basic definitions. The other parts introduce more specialized concepts needed in this thesis. Quantum mechanical systems are commonly described in terms of Hilbert spaces [166], which is why we start with revisiting some basic definitions of operators on Hilbert spaces in Section 3.1. In Section 3.2, we introduce the quantum mechanical description of the most basic experiments consisting of the preparation and measurement of a system. As an easy example, we discuss qudits and qubits in Section 3.3. Next, we consider composite systems in Section 3.4. We then establish the notion of quantum channels describing operations on quantum states in Section 3.5. For these first sections, we use [167] as a source. For our later discussion of recurrent structures, we define quantum channels with memory in Section 3.6. As we want to perform supervised learning on quantum states down the line, we need to define norms and distance measures on quantum states, which we will address in Section 3.7. In Section 3.8, we present the Haar measure as a way of sampling random unitaries and pure states. For tedious derivations in finite,

composite systems, we introduce tensor network notation in Section 3.9. We discuss QCs in Section 3.10. Last but not least, to set the stage for the remainder of the thesis, we highlight some results from the field of quantum machine learning in Section 3.11, setting an explicit focus on dissipative quantum neural networks (DQNNs).

3.1 Operators on Hilbert Spaces

A complex *Hilbert space* is a complex vector space $\mathcal{H}$ with a scalar product $\langle\cdot|\cdot\rangle : \mathcal{H}\times\mathcal{H} \to \mathbb{C}$ that is complete w.r.t. the norm $\|\cdot\|_{\mathcal{H}}^2 = \langle\cdot|\cdot\rangle$ induced by the scalar product. In the following, we assume all Hilbert spaces to be complex. As usual in quantum mechanics, we will use the convention that the scalar product is linear in the second argument, i.e.,

$$\langle\varphi|\lambda\psi\rangle = \lambda\langle\varphi|\psi\rangle = \langle\bar{\lambda}\varphi|\psi\rangle \quad \forall\ \varphi,\psi\in\mathcal{H}, \lambda\in\mathbb{C}.$$

Each Hilbert space has at least one *orthonormal basis (ONB)*, i.e., a labelled list of vectors $\{e_\mu\}_\mu \subset \mathcal{H}$ with $\langle e_\mu, e_\nu\rangle = \delta_{\mu\nu}$ and $\varphi = \sum_\mu \langle e_\mu,\varphi\rangle e_\mu\ \forall\varphi\in\mathcal{H}$. When we say *basis*, we usually mean an ONB. A Hilbert space with a countable basis is called *separable*. If not specified otherwise, we will assume that all infinite-dimensional Hilbert spaces are separable in the following. Let us now define bounded operators.

Definition and Proposition 3.1 (Bounded operators [167, 168])**.** *Let $\mathcal{H}_1, \mathcal{H}_2$ be Hilbert spaces, $A : \mathcal{H}_1 \to \mathcal{H}_2$ be linear. In shorthand, we write $A\psi := A(\psi)\ \forall\psi\in\mathcal{H}_1$. A is called a* bounded (linear) operator *iff there exists $c \geq 0$*

$$\|A\psi\|_{\mathcal{H}_2} \leq c\|\psi\|_{\mathcal{H}_1}\ \forall\psi\in\mathcal{H}_1.$$

We write $\mathcal{B}(\mathcal{H}_1,\mathcal{H}_2)$ for all bounded operators from $\mathcal{H}_1$ to $\mathcal{H}_2$ and $\mathcal{B}(\mathcal{H}_1)$ for all bounded operators from $\mathcal{H}_1$ to itself.
Let A be bounded. Then, we define its operator norm *by*

$$\|A\| = \sup_{\psi\in\mathcal{H}_1,\psi\neq 0} \frac{\|A\psi\|_{\mathcal{H}_2}}{\|\psi\|_{\mathcal{H}_1}}.$$

We define its adjoint $A^\dagger : \mathcal{H}_2 \to \mathcal{H}_1$ *by*

$$\langle\varphi|\, A^\dagger\psi\rangle_{\mathcal{H}_1} = \langle A\varphi\,|\psi\rangle_{\mathcal{H}_2} \quad \forall\varphi\in\mathcal{H}_1,\psi\in\mathcal{H}_2.$$

It is $\|A^\dagger\| = \|A\|$ and $(A^\dagger)^\dagger = A$. For $c\in\mathbb{C}$ and $B : \mathcal{H}_1 \to \mathcal{H}_2$ bounded, it is $(cA+B)^\dagger = \bar{c}A^\dagger + B^\dagger$.
We say A is an isometry *iff $\|A\varphi\|_{\mathcal{H}_2} = \|\varphi\|_{\mathcal{H}_1}$ for all $\varphi\in\mathcal{H}_1$ which is equivalent to $A^\dagger A = \mathbb{1}_{\mathcal{H}_1}, AA^\dagger = \mathbb{1}_{\mathcal{H}_2}$. An isometry $A : \mathcal{H}\to\mathcal{H}$ is called a* unitary*. We then have $A^\dagger A = AA^\dagger = \mathbb{1}_{\mathcal{H}}$.*
We call $A : \mathcal{H}\to\mathcal{H}$ Hermitian *or* self-adjoint *iff $A = A^\dagger$.*
A Hermitian operator $A\in\mathcal{B}(\mathcal{H})$ is called positive (semidefinite) *iff $\langle\varphi|\, A\varphi\rangle \geq 0\ \forall\varphi\in\mathcal{H}$. We then write $A\geq 0$. This is equivalent to all eigenvalues of A being greater or equal to 0. For $A,B\in\mathcal{B}(\mathcal{H})$, we write $A\geq B$ iff $A-B\geq 0$.*

Every finite-dimensional Hilbert space $\mathcal{H}$ is isomorph to $\mathbb{C}^d$ for a given $d\in\mathbb{N}$, i.e., there is an isometry mapping $\mathcal{H}$ to $\mathbb{C}^d$. This means that we can think of finite dimensional Hilbert spaces as $\mathbb{C}^d$ with the usual scalar product. In $\mathbb{C}^d$, bounded operators are given by matrices, and their adjoint is obtained by complex conjugation and transposition.

For $\varphi \in \mathcal{H}$ we define the maps

$$\begin{aligned} |\varphi\rangle : \mathbb{C} &\to \mathcal{H}, & \langle\varphi| : \mathcal{H} &\to \mathbb{C} \\ z &\mapsto |\varphi\rangle\, z = z\varphi & \psi &\mapsto \langle\varphi|\psi\rangle. \end{aligned}$$

Here, $\langle\cdot|$ is called a *bra* and $|\cdot\rangle$ is called a *ket* as their combination

$$(\langle\varphi| \circ |\psi\rangle)z = \langle\varphi|\,(z\psi) = z\,\langle\varphi|\,\psi\rangle \quad \forall\ \varphi,\ \psi \in \mathcal{H},\ z \in \mathbb{C}$$

is the scalar product, which was also called *bracket* by Dirac [169]. The bra is the adjoint of the ket, i.e.,

$$\langle\varphi| = |\varphi\rangle^\dagger \quad \forall\ \varphi \in \mathcal{H}.$$

For simplicity, in $\mathbb{C}^d$, we can think of kets as column vectors and the belonging bras as the complex conjugated row vectors obtained by transposing the column vector.

Other important concepts in QI, among others, are the *trace* of an operator, projections, the notion of convexity, and tensor products of vectors, Hilbert spaces, and operators.

Definition and Proposition 3.2 (Trace [170])**.** *Let $\mathcal{H}$ be a Hilbert space of dimension $d \in \mathbb{N} \cup \{\infty\}$, $A \in \mathcal{B}(\mathcal{H})$, and $\{\varphi_j\}_{j=1,\dots,d}$ an ONB of $\mathcal{H}$. We define*

$$\mathrm{tr}(A) = \sum_{j=1}^{d} \langle\phi_j|\, A\phi_j\rangle$$

to be the trace *of A. The trace of A is independent of the chosen ONB. If $\mathrm{tr}(|A|) < \infty$, where $|A| = \sqrt{A^\dagger A}$, we say A is in the* trace class *of $\mathcal{H}$ and write $A \in \mathcal{T}(\mathcal{H})$.*

Definition and Proposition 3.3 (Projections)**.** *Let $\mathcal{H}$ be a Hilbert space. An operator $P \in \mathcal{B}(\mathcal{H})$ is called a projection iff $P^2 = P$. We then write $P \in \mathcal{P}(\mathcal{H})$. Operators of the form $P = |\eta\rangle\,\langle\eta|$, where $\eta \in \mathcal{H}$ is a unit vector, are projections and are called* one-dimensional projections*.*

Definition 3.1 (Convexity [171])**.** *A subset S of a real or complex vector space V is called* convex *iff $tf + (1-t)g \in S$ for all $t \in [0,1]$, $f, g \in S$. Let $h \in S$. If there exist $t \in (0,1)$, $f, g \in S$ such that $h = tf + (1-t)g$, we say h is an* interior point *of S. If not, we say h is an* extremal point *of S.*

Definition and Proposition 3.4 (Tensor product)**.** *Let $\mathcal{H}_A$ and $\mathcal{H}_B$ be Hilbert spaces. We can define a Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, called* tensor product *of $\mathcal{H}_A$ and $\mathcal{H}_B$, by the completion of the span of $\{\psi \otimes \varphi \,|\, \psi \in \mathcal{H}_A,\ \varphi \in \mathcal{H}_B\}$, where $\psi \otimes \varphi$ is linear w.r.t. both arguments, i.e.,*

$$\begin{aligned} c(\psi \otimes \varphi) &= (c\psi) \otimes \varphi = \psi \otimes (c\varphi) \\ (\psi_1 + \psi_2) \otimes \varphi &= \psi_1 \otimes \varphi + \psi_2 \otimes \varphi \\ \psi \otimes (\varphi_1 + \varphi_2) &= \psi \otimes \varphi_1 + \psi \otimes \varphi_2 \end{aligned}$$

where $\psi, \psi_1, \psi_2 \in \mathcal{H}_A, \varphi, \varphi_1, \varphi_2 \in \mathcal{H}_B, c \in \mathbb{C}$. On $\mathcal{H}_A \otimes \mathcal{H}_B$, we define addition of two elements $v = \sum_i \psi_i \otimes \varphi_i \in \mathcal{H}_A \otimes \mathcal{H}_B$ and $w = \sum_j \psi_j \otimes \varphi_j \in \mathcal{H}_A \otimes \mathcal{H}_B$ by $v + w = \sum_i \psi_i \otimes \varphi_i + \sum_j \psi_j \otimes \varphi_j$, and their scalar product by $\langle v\,|w\rangle = \sum_i \sum_j \langle\psi_i\,|\psi_j\rangle_A\,\langle\varphi_i\,|\varphi_j\rangle_B$.

Given ONBs $\{\psi_i\}_i$ and $\{\varphi_j\}_j$ of $\mathcal{H}_A$ and $\mathcal{H}_B$, $\{\psi_i \otimes \varphi_j\}_{ij}$ forms an ONB of $\mathcal{H}_A \otimes \mathcal{H}_B$.

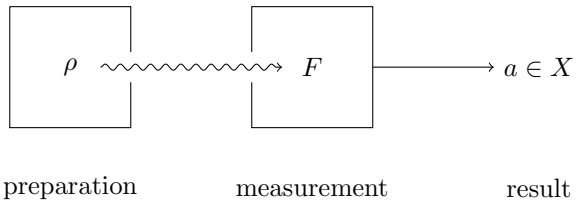


Figure 3.1: **Preparation and Measurement.** We prepare the state $\rho \in \mathcal{D}(\mathcal{H})$ and perform a measurement using an observable $F = \{F_a\}_{a \in X}$ on $\mathcal{H}$. With probability $p_a = \mathrm{tr}(\rho F_a)$, we get the measurement result $a \in X$.

Let $A \in \mathcal{B}(\mathcal{H}_A)$ and $B \in (\mathcal{H}_B)$. We define $A \otimes B \in \mathcal{B}(\mathcal{H}_A \otimes \mathcal{H}_B)$ by

$$(A \otimes B)(\psi \otimes \varphi) = A\psi \otimes B\varphi, \quad \forall\ \psi \in \mathcal{H}_A,\ \varphi \in \mathcal{H}_B$$

and its linear extension.

The tensor product of two spaces $\mathbb{C}^{d_1}$ and $\mathbb{C}^{d_2}$ is isomorph to $\mathbb{C}^{d_1 d_2}$, and the tensor product between operators is simply given by the Kronecker product.

Example 3.1 (Kronecker product [172])**.** *For two matrices $A \in \mathcal{B}(\mathbb{C}^{d_1}, \mathbb{C}^{d_2}), B \in \mathcal{B}(\mathbb{C}^{d_3}, \mathbb{C}^{d_4})$, their* Kronecker product *is defined as*

$$A \otimes B = (a_{ij}B)_{i=1,\ldots,d_1;\, j=1,\ldots,d_2}$$

which is in $\mathcal{B}(\mathbb{C}^{d_1 d_3}, \mathbb{C}^{d_2 d_4})$. Hence, for

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad \in \mathcal{B}(\mathbb{C}^2, \mathbb{C}^2)$$

it is

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}.$$

3.2 Preparation and Measurement

At its core, quantum mechanics is a probabilistic theory. That means that we generally cannot predict the measurement result of an individual measurement, but only probability distributions over measurement results. Most physical experiments can be divided into a preparation and a measurement phase, as depicted in Figure 3.1.

3.2.1 Preparation

We collect all information on the preparation in a *quantum state* or *density operator* defined as follows.

Definition and Proposition 3.5 (Density operator)**.** *A* density operator *on a Hilbert space $\mathcal{H}$ is a positive operator $\rho \in \mathcal{T}(\mathcal{H})$ with $\mathrm{tr}\rho = 1$. We write*

$$\mathcal{D}(\mathcal{H}) := \{\rho \in \mathcal{T}(\mathcal{H}) | \rho \geq 0,\ \mathrm{tr}\rho = 1\}$$

for the set of all density operators on $\mathcal{H}$. It is convex, and its extremal points are given by one-dimensional projections, i.e., they are of the form $\rho = |\psi\rangle\langle\psi|$ for a unit vector $\psi \in \mathcal{H}$, and called pure states*. Sometimes, we then also call $|\psi\rangle$ a state. All other states are called mixed states.*

3.2.2 Measurement

The measurements are in all features commonly described by *positive operator valued measurementss (POVMs)*. Essentially, we associate a set of operators with a measurement, one for each possible measurement result, and write the probability distribution as the trace of the state times the different operators, as formalized in Definition and Proposition 3.6.

Definition and Proposition 3.6 (POVM)**.** *An operator $F \in \mathcal{B}(\mathcal{H})$ on a Hilbert space $\mathcal{H}$ with $0 \leq F \leq \mathbb{1}$ is called an* effect*. The set of effects is convex, and its extremal points are given by the projections.*
Given a countable[1] *set of possible measurement results X, we call a collection of effects $F = \{F_a\}_{a\in X}$ with $\sum_{a\in X} F_a = \mathbb{1}$ a* positive operator valued measurements (POVM) *or* observable*. The probability of measuring $a \in X$ can then be written as*

$$p(a) = \mathrm{tr}(\rho F_a).$$

We need these restrictions on the state and operator space so that for every state ρ and every observable F, the distribution defined by $p(a) = \mathrm{tr}(\rho F_a)$ is a proper probability distribution.

3.3 Qudits

Every Hilbert space of finite dimension d is isomorph to $\mathbb{C}^d$. The simplest non-trivial Hilbert space is $\mathcal{H} = \mathbb{C}^2$. Its standard basis vectors are given by

$$e_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } e_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

and we often write $|0\rangle = |e_0\rangle$ and $|1\rangle = |e_1\rangle$. The states on $\mathbb{C}^2$ are called *qubits* and are usually considered the quantum informational analog to the classical bits 0 and 1. More generally, states on $\mathbb{C}^d$ are called *qudits*. The space of operators is then given by the $d \times d$ matrices. On $\mathcal{H} = \mathbb{C}^2$,[2] we can write every operator in terms of the identity $\sigma_0 = \mathbb{1}$ and the traceless and Hermitian *Pauli operators*

$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \ \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \ \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Every state $\rho \in \mathcal{D}(\mathbb{C}^2)$ can be written as $\rho = \frac{1}{2}\mathbb{1} + \vec{r}\cdot\vec{\sigma}$ with $\vec{r} \in \mathbb{R}^3, |\vec{r}| \leq 1$ and $\vec{\sigma} = (\sigma_1, \sigma_2, \sigma_3)$. We can then depict the set of density operators on $\mathbb{C}^2$ in $\mathbb{R}^3$ as the sphere with $|\vec{r}| \leq 1$. This is shown in Figure 3.2 and is often called the *Bloch sphere*. The pure states can be found on the surface of the sphere, and the mixed states can be found in its interior.

[1] If X is not countable, the sum changes to an integral.
[2] We can do a similar construction for $\mathcal{H} = \mathbb{C}^d$ but will leave it out for simplicity.

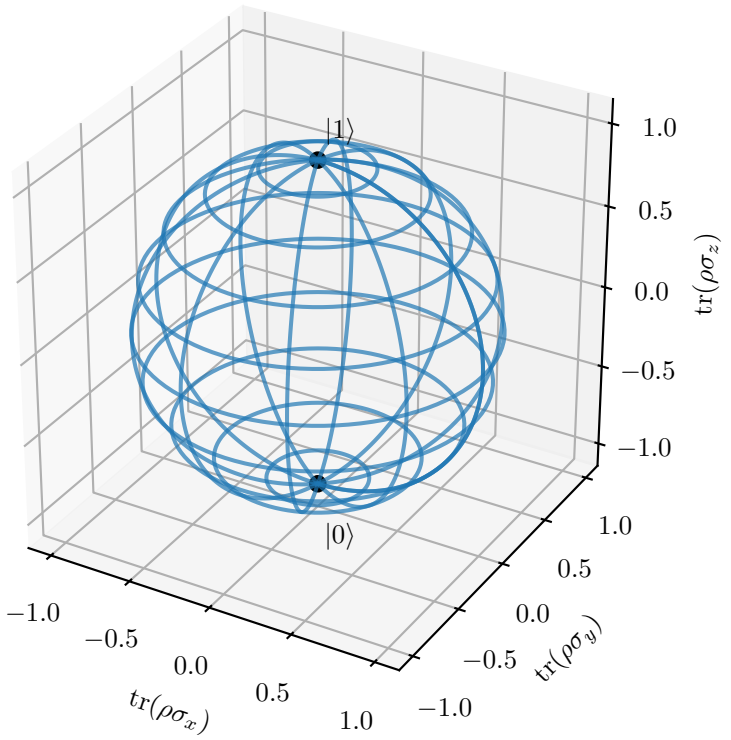


Figure 3.2: **Bloch sphere.** Quantum states on $\mathbb{C}^2$ can be depicted in a coordinate system with axes $\mathrm{tr}(\sigma_i\rho)$, $i = 1, 2, 3$. In this coordinate system, the pure states are on the surface of the sphere with radius 1, and the mixed states are in its interior. The states $|0\rangle$ and $|1\rangle$ can be found on its south and north poles, respectively.

3.4 Composite Systems

Consider two (distinguishable) systems, A and B, described by Hilbert spaces $\mathcal{H}_\mathrm{A}$ and $\mathcal{H}_\mathrm{B}$. The Hilbert space corresponding to the composite system AB is then given by their tensor product $\mathcal{H}_\mathrm{AB} = \mathcal{H}_\mathrm{A} \otimes \mathcal{H}_\mathrm{B}$. We then say A and B are subsystems of AB. As not all operators in $\mathcal{B}(\mathcal{H}_\mathrm{AB})$ can be written as $A \otimes B$ with $A \in \mathcal{H}_\mathrm{A}, B \in \mathcal{H}_\mathrm{B}$, we introduce the notion of entanglement.

Definition 3.2 (Entanglement)**.** *Let $\mathcal{H}_A$, $\mathcal{H}_B$ be Hilbert spaces, $\mathcal{H}_{AB} = \mathcal{H}_A \otimes \mathcal{H}_B$. We say the state $\rho \in \mathcal{D}(\mathcal{H}_{AB})$ is*

- *a* product *state iff $\rho = \rho_A \otimes \rho_B$ for $\rho_A \in \mathcal{D}(\mathcal{H}_A), \rho_B \in \mathcal{D}(\mathcal{H}_B)$. We then write $\rho \in \mathcal{D}^{fac}(\mathcal{H}_{AB})$.*
- *a* separable *state iff $\rho = \sum_{\lambda=1}^{n} p_\lambda \rho_{A,\lambda} \otimes \rho_{B,\lambda}$ for $\rho_{A,\lambda} \in \mathcal{D}(\mathcal{H}_A), \rho_{B,\lambda} \in \mathcal{D}(\mathcal{H}_B)$, $p_\lambda \geq 0, \sum_{\lambda=1}^{n} p_\lambda = 1, n \in \mathbb{N}$. We then write $\rho \in \mathcal{D}^{sep}(\mathcal{H}_{AB})$.*
- *an* entangled *state iff $\rho \notin \mathcal{D}^{sep}(\mathcal{H}_{AB})$.*

All separable pure states are also factorized.

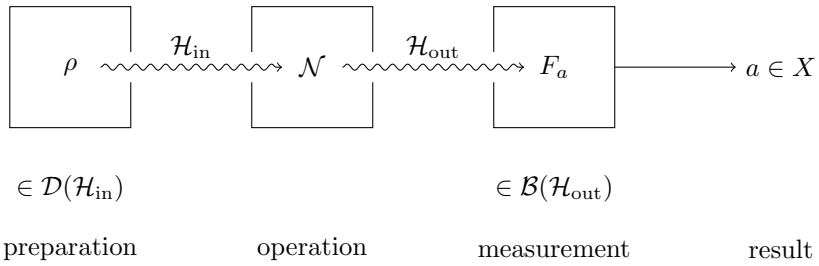


Figure 3.3: **Channel.** First, we prepare a state $\rho \in \mathcal{D}_{\mathcal{H}_{\text{in}}}$. We then use an operation $\mathcal{N}$, changing the Hilbert space describing the system from $\mathcal{H}_{\text{in}}$ to $\mathcal{H}_{\text{out}}$. Lastly, we measure the state by a POVM $F = \{F_a\}_{a \in X}$ providing the measured value $a \in X$.

Assume we are given a bipartite system with Hilbert space $\mathcal{H}_{\text{AB}} = \mathcal{H}_{\text{A}} \otimes \mathcal{H}_{\text{B}}$ but we only have access to one subsystem, w.l.o.g. system A, i.e., we can only operate on $\mathcal{H}_{\text{A}}$. Given a *joint state* $\rho_{\text{AB}} \in \mathcal{D}(\mathcal{H}_{\text{AB}})$, we might be interested in the measurement statistics if we only measure subsystem A and ignore subsystem B. This information is given by the *reduced state* on system A

$$\rho_{\text{A}} = \text{tr}_{\text{B}}(\rho_{\text{AB}})$$

where tr_{B} is the *partial trace* w.r.t. subsystem B that is defined as follows. We also say we *trace out* subsystem B.

Definition and Proposition 3.7 (Partial trace)**.** *Let $\mathcal{H}_A$, $\mathcal{H}_B$ be Hilbert spaces, and $\mathcal{H}_{AB} = \mathcal{H}_A \otimes \mathcal{H}_B$. We define the* partial trace *over subsystem A as the map*

$$\text{tr}_A : \mathcal{T}(\mathcal{H}_{AB}) \to \mathcal{T}(\mathcal{H}_B)$$

that fulfills

$$\text{tr}\left(\text{tr}_A\left(T\right) E\right) = \text{tr}\left(T\left(E \otimes \mathbb{1}_B\right)\right) \quad \forall\, T \in \mathcal{T}(\mathcal{H}_{AB}),\ E \in \mathcal{B}(\mathcal{H}_B).$$

Given ONBs $\{\psi_i\}$ of $\mathcal{H}_A$ and $\{\varphi_m\}$ of $\mathcal{H}_B$, tr_A can be written as

$$\text{tr}_A(T) = \sum_{j,n,m} \langle \psi_j \otimes \varphi_m | T | \psi_j \otimes \varphi_n \rangle | \varphi_m \rangle \langle \varphi_n | = \sum_j \left(\langle \psi_j | \otimes \mathbb{1}_B\right) T \left(| \psi_j \rangle \otimes \mathbb{1}_B\right)$$

fo all $T \in \mathcal{T}(\mathcal{H}_{AB})$. The partial trace is independent of the used ONB and is defined analogously on subsystem B.

If the reduced states are pure, the joint state is a product state.

3.5 Quantum Channels

Up until now, we discussed experiments split into a preparation and a measurement phase. This is a coarse picture as, for many experiments, either the measurement, the preparation, or both can be split into several phases. This introduces the notion of an operation, as depicted in Figure 3.3.

This operation can be seen either as part of the preparation phase or the measurement phase, which is called *Schrödinger* or *Heisenberg picture*, respectively, and shown in Figure 3.4. In

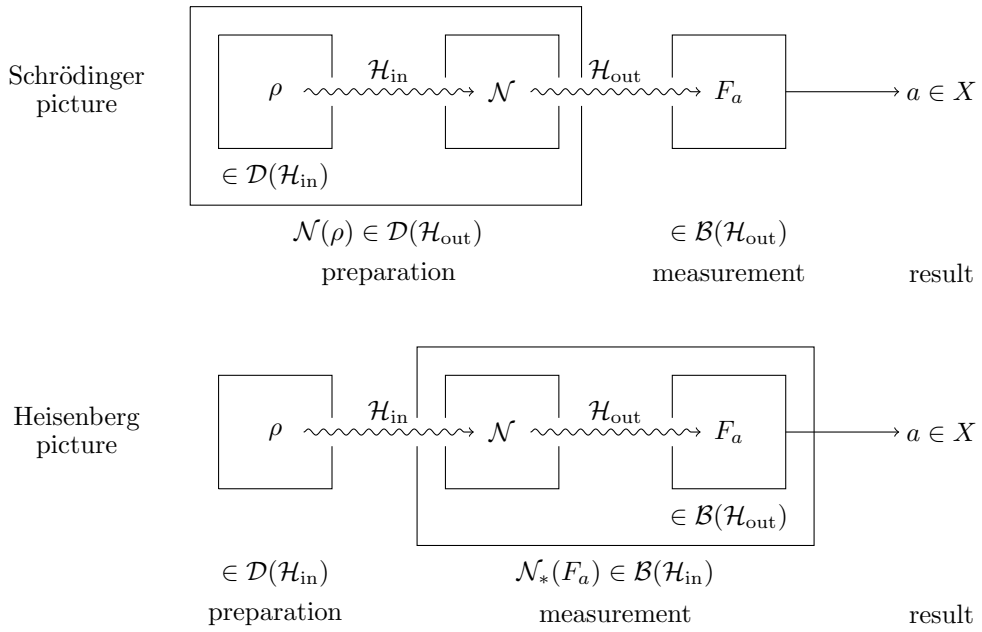


Figure 3.4: **Schrödinger vs. Heisenberg picture.** In the Schrödinger picture, we see the evolution as part of the preparation. After preparing the state $\rho \in \mathcal{D}(\mathcal{H}_{\text{in}})$, we apply the evolution $\mathcal{N}$ to get the state $\mathcal{N}(\rho) \in \mathcal{D}(\mathcal{H}_{\text{out}})$. This state is then measured by a POVM $F = \{F_a\}_{a \in X}$ on $\mathcal{H}_{\text{out}}$ providing the result $a \in X$ with probability $p_a = \operatorname{tr}(\mathcal{N}(\rho)F_a)$. In the Heisenberg picture, we see the evolution as part of the measurement. We prepare the state $\rho \in \mathcal{D}(\mathcal{H}_{\text{in}})$. The evolution $\mathcal{N}$ and the measurement given by the POVM $F = \{F_a\}_{a \in X}$ on $\mathcal{H}_{\text{out}}$ are reduced to a measurement $\{\mathcal{N}_*(F_a)\}_{a \in X}$ providing results $a \in X$ with probability $p_a = \operatorname{tr}(\rho \mathcal{N}_*(F_a))$.

the Schrödinger picture, we regard the operation $\mathcal{N}$ as a map $\mathcal{N} : \mathcal{T}(\mathcal{H}_{\text{in}}) \to \mathcal{T}(\mathcal{H}_{\text{out}})$ mapping density operators again to density operators. In the Heisenberg picture, we regard it as a map $\mathcal{N}_* : \mathcal{B}(\mathcal{H}_{\text{out}}) \to \mathcal{B}(\mathcal{H}_{\text{in}})$ mapping POVMs again to POVMs. As the output probabilities have to be the same after a measurement, regardless of our description in the Schrödinger or Heisenberg picture, $\mathcal{N}_*$ is the Banach space adjoint to $\mathcal{N}$. We will focus on the Schrödinger picture here.

We have to make sure that $\mathcal{N}$ correctly maps density operators to density operators. This means that $\mathcal{N}$ should preserve the trace and positivity of its input. In fact, this should not only be true in the system $\mathcal{N}$ is acting on but also if it only acts on a subsystem of a composite system, i.e., when there is an *innocent bystander* as depicted in Figure 3.5. This leads us to the definition of positivity and complete positivity.

Definition 3.3 (Positivity and Complete Positivity)**.** *Let $\mathcal{H}_{\text{in}}$, $\mathcal{H}_{\text{out}}$ be Hilbert spaces, and $\mathcal{N} : \mathcal{B}(\mathcal{H}_{\text{in}}) \to \mathcal{B}(\mathcal{H}_{\text{out}})$. We say $\mathcal{N}$ is* positive *iff $\mathcal{N}(A) \geq 0$ for all $A \in \mathcal{B}(\mathcal{H}_{\text{in}}), A \geq 0$. We say $\mathcal{N}$ is* completely positive *iff*

$$(\mathcal{N} \otimes \mathbb{1}_n)(A) \geq 0 \quad \forall\, n \in \mathbb{N},\ A \in \mathcal{B}(\mathcal{H}_{\text{in}} \otimes \mathbb{C}^n),\ A \geq 0$$

where $\mathbb{1}_n : \mathcal{B}(\mathbb{C}^n) \to \mathcal{B}(\mathbb{C}^n)$ is the identity.

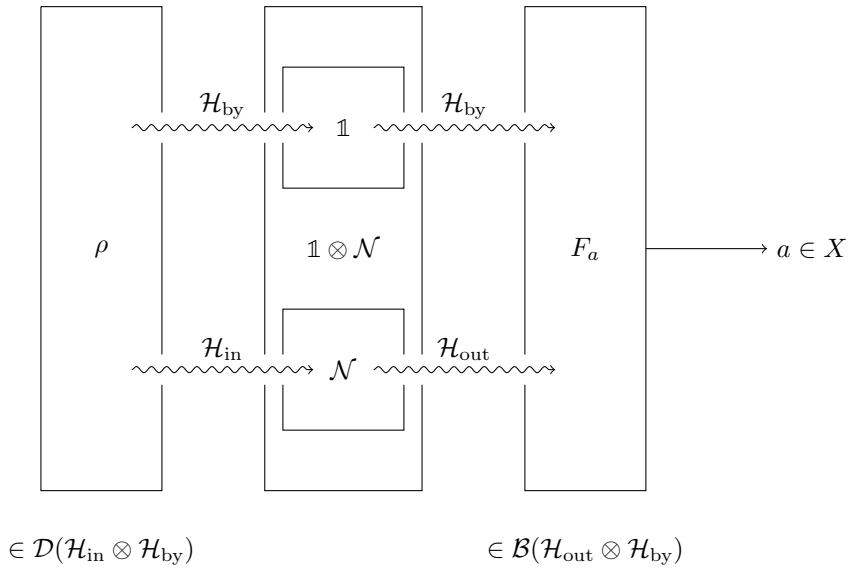


Figure 3.5: Operation with an innocent bystander - A state $\rho \in \mathcal{D}(\mathcal{H}_{\text{by}} \otimes \mathcal{H}_{\text{in}})$ is prepared. Then, the evolution $\mathcal{N}$ is applied to the subsystem with Hilbert space $\mathcal{H}_{\text{in}}$. The state is then measured by the POVM $F = \{F_a\}_{a \in X}$ on $\mathcal{H}_{\text{by}} \otimes \mathcal{H}_{\text{out}}$ providing the result $a \in X$.

Not every positive operation is also completely positive. We can now define a *channel*, the most general map $\mathcal{N} : \mathcal{T}(\mathcal{H}_{\text{in}}) \to \mathcal{T}(\mathcal{H}_{\text{out}})$, mapping density operators to density operators without classical in- or outputs.

Definition 3.4 (Channel)**.** *Let $\mathcal{H}_{\text{in}}$, $\mathcal{H}_{\text{out}}$ be Hilbert spaces, $\mathcal{N} : \mathcal{T}(\mathcal{H}_{\text{in}}) \to \mathcal{T}(\mathcal{H}_{\text{out}})$. We say $\mathcal{N}$ is a* channel *iff it is linear, completely positive, and* trace preserving*, i.e.,*

$$\text{tr}(\mathcal{N}(A)) = \text{tr}(A) \quad \forall\ A \in \mathcal{T}(\mathcal{H}_{\text{in}}).$$

The easiest example of a channel is a unitary channel.

Example 3.2 (Unitary Channel)**.** *Let $\mathcal{H}$ be a Hilbert space. For every unitary $U \in \mathcal{B}(\mathcal{H})$, the map $\mathcal{N}_U : \mathcal{T}(\mathcal{H}) \to \mathcal{T}(\mathcal{H})$,*

$$\mathcal{N}_U(X) = UXU^\dagger \quad \forall\ X \in \mathcal{B}(\mathcal{H})$$

is a channel. We call channels of this form unitary channels*.*

A special case of this is the time evolution of states.

Example 3.3 (Time evolution [51])**.** *In quantum mechanics, the time evolution of states in an isolated system described by the Hilbert space $\mathcal{H}$ is governed by the* Hamiltonian*. The Hamiltonian is a Hermitian operator on said Hilbert space. If the Hamiltonian H is independent of the time, the time evolution of states is then governed by*

$$\rho(t_2) = U(t_1, t_2)\rho(t_1)U(t_1, t_2)^\dagger$$

where

$$U(t_1, t_2) = \exp\left(\frac{-iH}{\hbar}(t_2 - t_1)\right)$$

is the time evolution operator *and $\hbar$ is Planck's constant.*

Another example is the partial trace. In fact, we can write every channel as a combination of tensoring to a pure state, using a unitary and a partial trace. This is formalized in Stinespring's theorem for channels.

Proposition 3.1 (Stinespring's theorem for channels)**.** *Let $\mathcal{H}$ be a Hilbert space. The map $\mathcal{N} : \mathcal{T}(\mathcal{H}) \to \mathcal{T}(\mathcal{H})$ is a channel iff there exists a Hilbert space $\mathcal{H}_E$, a pure state $\tau \in \mathcal{D}(\mathcal{H}_E)$, and a unitary $U \in \mathcal{B}(\mathcal{H} \otimes \mathcal{H}_E)$ such that*

$$\mathcal{N}(\rho) = \mathrm{tr}_E(U(\rho \otimes \tau)U^\dagger) \quad \forall\ \rho \in \mathcal{D}(\mathcal{H}).$$

Another way to write down channels is via their Kraus decomposition.

Proposition 3.2 (Kraus decomposition)**.** *Let $\mathcal{H}$ be a Hilbert space. The map $\mathcal{N} : \mathcal{T}(\mathcal{H}) \to \mathcal{T}(\mathcal{H})$ is a channel iff there exists a sequence of operators $\{K_j\}_j$, called* Kraus operators, *such that*

$$\mathcal{N}(A) = \sum_j K_j A K_j^\dagger \quad \forall\ A \in \mathcal{B}(\mathcal{H}), \quad \sum_j K_j^\dagger K_j = \mathbb{1}, \quad K_j \in \mathcal{B}(\mathcal{H}).$$

If $\dim \mathcal{H} < \infty$, we can choose a finite number of Kraus operators.

3.6 Quantum Channels with Memory

A *causal quantum automaton* is a map on quantum states, for which outputs at time t only depend on inputs at times $t' \leq t$. All such maps can be written as a concatenated quantum channel with memory preceded by an initialization of the memory [90].

Definition 3.5 (Quantum channels with memory [90])**.** *Let $\mathcal{H}^{\mathrm{in}}$, $\mathcal{H}^{\mathrm{out}}$ and $\mathcal{H}^{\mathrm{mem}}$ be Hilbert spaces. A* quantum channel with memory *is represented by a quantum channel $\mathcal{N} : \mathcal{T}(\mathcal{H}^{\mathrm{mem}}) \otimes \mathcal{T}(\mathcal{H}^{\mathrm{in}}) \to \mathcal{T}(\mathcal{H}^{\mathrm{out}}) \otimes \mathcal{T}(\mathcal{H}^{\mathrm{mem}})$. We call $\mathcal{H}^{\mathrm{mem}}$ the* memory, *$\mathcal{H}^{\mathrm{in}}$ the input, and $\mathcal{H}^{\mathrm{out}}$ the output Hilbert space. An input sequence of length $N \in \mathbb{N}$ is processed by the N-times concatenated channel $\mathcal{M}_N : \mathcal{T}(\mathcal{H}^{\mathrm{mem}}) \otimes \mathcal{T}(\mathcal{H}^{\mathrm{in}})^{\otimes N} \to \mathcal{T}(\mathcal{H}^{\mathrm{out}})^{\otimes N} \otimes \mathcal{T}(\mathcal{H}^{\mathrm{mem}})$ given by*

$$\mathcal{M}_N = \left(\mathbb{1}^{\mathrm{out}}_{1,\ldots,N-1} \otimes \mathcal{N}\right) \circ \cdots \circ \left(\mathbb{1}^{\mathrm{out}}_1 \otimes \mathcal{N} \otimes \mathbb{1}^{\mathrm{in}}_{3,..,N}\right) \circ \left(\mathcal{N} \otimes \mathbb{1}^{\mathrm{in}}_{2,..,N}\right).$$

In the same way as for classical RNNs (see Section 2.1.4), we can again depict a quantum channel with memory in a compressed version and an unfolded version. Both versions are shown in Figure 3.6.

An example of such a quantum channel with memory is the SWAP or delay channel.

Example 3.4 (SWAP/delay channel [173, 174])**.** *Now choose $\mathcal{H}^{\mathrm{in}} = \mathcal{H}^{\mathrm{out}} = \mathcal{H}$, $\mathcal{H}^{\mathrm{mem}} = \mathcal{H}^{\otimes k}$, and define the memory channel $\mathcal{N}_{\text{delay by } k} : \mathcal{B}(\mathcal{H}^{\mathrm{mem}}) \otimes \mathcal{B}(\mathcal{H}^{\mathrm{in}}) \to \mathcal{B}(\mathcal{H}^{\mathrm{out}}) \otimes \mathcal{B}(\mathcal{H}^{\mathrm{mem}})$ by*

$$\begin{aligned}\mathcal{N}_{\text{delay by } k}(\rho^{\mathrm{mem},1} \otimes \cdots \otimes \rho^{\mathrm{mem},k} \otimes \rho^{\mathrm{in}}) &= \rho^{\mathrm{mem},1} \otimes \cdots \otimes \rho^{\mathrm{mem},k} \otimes \rho^{\mathrm{in}} \\ &=: \rho^{\mathrm{out}} \otimes \tilde{\rho}^{\mathrm{mem},1} \otimes \cdots \otimes \tilde{\rho}^{\mathrm{mem},k}.\end{aligned}$$

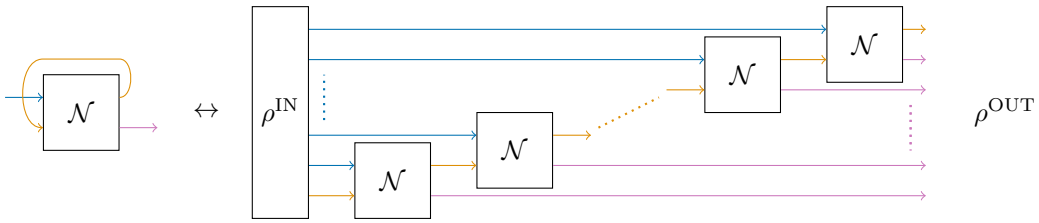


Figure 3.6: **Quantum channel with memory.** General structure of a quantum channel with memory with underlying channel $\mathcal{N}$. The left-hand side shows the compressed version and the right-hand side shows the same unfolded version with N iterations. The purple lines are registers with Hilbert space $\mathcal{H}^{\mathrm{out}}$, the orange ones registers with Hilbert space $\mathcal{H}^{\mathrm{mem}}$, and the blue ones registers with Hilbert space $\mathcal{H}^{\mathrm{in}}$. Given an input state ρ^{IN} on $\mathcal{H}^{\mathrm{IN}} = \mathcal{H}^{\mathrm{mem}} \otimes \mathcal{H}^{\mathrm{in}^{\otimes N}}$, the output is a state ρ^{OUT} on $\mathcal{H}^{\mathrm{out}^{\otimes N}} \otimes \mathcal{H}^{\mathrm{mem}}$.

Hence, the output is given by $\rho^{\mathrm{mem},1}$*, and the new memory state is* $\rho^{\mathrm{mem},2} \otimes \cdots \otimes \rho^{\mathrm{mem},k} \otimes \rho^{\mathrm{in}}$*. This channel can be achieved by a number of SWAPs in the memory register and between the memory and the input/output register. Given* N *consecutive inputs* ρ_x^{in}*, and assuming the memory was initialized in a product state* $\rho^{\mathrm{mem},1} \otimes \cdots \otimes \rho^{\mathrm{mem},k}$*, i.e.,*

$$\rho^{\mathrm{IN}} = \rho^{\mathrm{mem},1} \otimes \cdots \otimes \rho^{\mathrm{mem},k} \otimes \rho_1^{\mathrm{in}} \otimes \cdots \otimes \rho_N^{\mathrm{in}},$$

after applying the N*-times concatenated channel, we get the output*

$$\begin{aligned} \rho^{\mathrm{OUT}} = \mathcal{M}_N(\rho^{\mathrm{IN}}) &= \rho^{\mathrm{mem},1} \otimes \cdots \otimes \rho^{\mathrm{mem},k} \otimes \rho_1^{\mathrm{in}} \otimes \cdots \otimes \rho_N^{\mathrm{in}} \\ &=: \rho_1^{\mathrm{out}} \otimes \cdots \otimes \rho_N^{\mathrm{out}} \otimes \tilde{\rho}^{\mathrm{mem},1} \otimes \cdots \otimes \tilde{\rho}^{\mathrm{mem},k}. \end{aligned}$$

Hence, the x^{th} *output is given by*

$$\begin{aligned} \rho_x^{\mathrm{out}} &= \rho^{\mathrm{mem},x}, \ x = 1, ..., k \\ \rho_x^{\mathrm{out}} &= \rho_{x-k}^{\mathrm{in}}, \ x = k+1, ..., N. \end{aligned}$$

This means the channel essentially delays the output of each input ρ_x^{in} *by* k *time steps.*

3.7 Norms and Fidelity

For many ML applications, we need some notion of distance between data points, and in the case of data consisting of quantum states, between quantum states. In more detail, we want to consider different norms on $\mathcal{B}(\mathcal{H})$ for a Hilbert space $\mathcal{H}$ and the fidelity between states.

An already introduced norm in Definition and Proposition 3.1 is the operator norm of an operator $A \in \mathcal{B}(\mathcal{H})$. A family of norms on $\mathcal{B}(\mathcal{H})$ including the operator norm, called the *Schatten-p-norms*, is defined as

$$\|A\|_p = \left(\mathrm{tr}\left(|A|\right)^p\right)^{\frac{1}{p}} \tag{3.7.1}$$

for $1 \leq p < \infty$, where $|A| = \left(A^\dagger A\right)^{\frac{1}{2}} = \left(AA^\dagger\right)^{\frac{1}{2}}$, and

$$\|A\|_\infty = \|A\|$$

for all $A \in \mathcal{B}(\mathcal{H})$ [175]. The Schatten-2 norm is also called the *Hilbert-Schmidt norm*, and the Schatten-1 norm is called the *trace norm*. When we compare two quantum states $\rho_1, \rho_2 \in \mathcal{D}(\mathcal{H})$ via the trace distance, we get

$$\frac{1}{2}\|\rho_1 - \rho_2\|_1 = \sup_{F \in \mathcal{B}(\mathcal{H}), 0 \leq F \leq \mathbb{1}} |\mathrm{tr}(\rho_1 F) - \mathrm{tr}(\rho_2 F)|.$$

This is the largest difference in comparing the probability distributions after measuring the two states.

The most used criterion for the closeness of two states, however, is the *fidelity*. The fidelity of two states $\rho_1, \rho_2 \in \mathcal{D}(\mathcal{H})$ is defined as[3]

$$F(\rho_1, \rho_2) = \left(\mathrm{tr}\left((\sqrt{\rho_1}\rho_2\sqrt{\rho_1})^{1/2}\right)\right)^2.$$

It takes on values between 0 and 1 and is optimal, i.e. $\rho_1 = \rho_2$, at 1. If one of the two states is pure, the fidelity simplifies to

$$F\left(|\psi\rangle\langle\psi|, \rho\right) = \langle\psi|\rho|\psi\rangle = \mathrm{tr}\left(|\psi\rangle\langle\psi|\rho\right). \tag{3.7.2}$$

We can see that it is the overlap between the two states [176]. The fidelity and trace distance relate via

$$\|\rho_1 - \rho_2\|_1 \leq \sqrt{1 - F(\rho_1, \rho_2)}$$

with equality for pure states, which makes the fidelity often the slightly stronger criterion [51].

3.8 Sampling Unitaries and Pure States

We later need a way to sample random unitaries and quantum states, which we will do w.r.t. the *Haar measure*. The Haar measure is a unique probability measure that can generally be defined on locally compact topological groups, such as the unitary group [177] on $\mathbb{C}^n$

$$\mathcal{U}(n) = \{U \in \mathcal{B}(\mathbb{C}^n) \mid \; UU^* = U^*U = \mathbb{1}_n\},$$

and is left-invariant w.r.t. multiplications with elements of the group [178]. We can then use the Haar measure on unitaries not only to sample unitaries but also to sample pure states. To sample a pure state $|\psi\rangle$, we fix a pure state, e.g., $|0\rangle$, sample a unitary U w.r.t. the Haar measure and then set $|\psi\rangle = U|0\rangle$. This gives us a unique uniform probability measure on pure states that is invariant to left multiplication with a unitary [179–181].

3.9 Tensor Networks

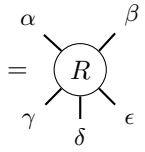
This section follows [182] if not mentioned otherwise.

In many quantum algorithms, we deal with operations on Hilbert spaces of the form $\mathbb{C}^{d_1} \otimes \mathbb{C}^{d_2} \otimes \ldots \otimes \mathbb{C}^{d_r} \cong \mathbb{C}^{d_1 \times d_2 \times \ldots \times d_r}$. Elements of these vector spaces are called *rank-r tensor*. A rank-0 tensor is simply a scalar, a rank-1 tensor is a vector, and a rank-2 tensor is a matrix. Tensor network notation (TNN) can be used to simplify calculations with tensors and get a visual representation of tensor calculations.

[3] It is also commonly defined as the square root of that to fit the classical fidelity [51], but we stick with this definition here to simplify the optimization problems later.

3.9.1 Tensor

A tensor of rank r is simply represented by a geometrical shape with r legs. For example, a rank 5 tensor $R = \left(R^{\alpha\beta}_{\gamma\delta\epsilon}\right)$ can be written as

$R^{\alpha\beta}_{\gamma\delta\epsilon} =$ 

or $R =$ 

.

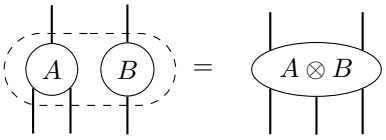
In some cases, the color, shape, and direction of the body or legs can have a meaning. For quantum states, kets and bras usually point their legs in different directions.

3.9.2 Tensor Product

A tensor product of a rank-r tensor A and a rank-s tensor B is given as

$$(A \otimes B)_{i_1,\ldots,i_r,j_1,\ldots,j_s} = A_{i_1,\ldots,i_r} \cdot B_{j_1,\ldots,j_s}.$$

In TNN, we e.g. write



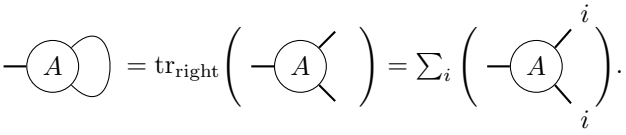
for $r = 3, s = 2$.

3.9.3 Trace

A (partial) trace over index m and n with the same dimension $d = d_m = d_n$ of a rank-r tensor A is given by

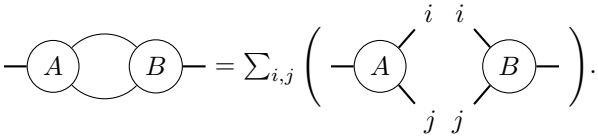
$$(\mathrm{tr}_{m,n}(A))_{i_1,\ldots,i_{m-1},i_{m+1},\ldots,i_{n-1},i_{n+1},\ldots,i_r} = \sum_{\alpha=1}^{d} A_{i_1,\ldots,i_{m-1},\alpha,i_{m+1},\ldots,i_{n-1},\alpha,i_{n+1},\ldots,i_r}.$$

In TNN, we, e.g., for a rank-3 tensor A, write

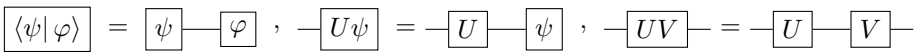


3.9.4 Contraction

Another much-used operation is the contraction of two tensors, corresponding to first tensoring the two and then tracing out the indices between the two tensors. We write, e.g.,



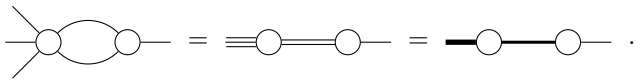
Standard examples for this are the scalar product of two vectors $\psi, \varphi \in \mathbb{C}^d$, the multiplication of a vector $\psi \in \mathbb{C}^{d_2}$ and a matrix $U \in \mathbb{C}^{d_1 \times d_2}$, and matrix multiplication of two matrices $U \in \mathbb{C}^{d_1 \times d_2}$, $V \in \mathbb{C}^{d_2 \times d_3}$, which we write as



in TNN.

3.9.5 Grouping and Splitting

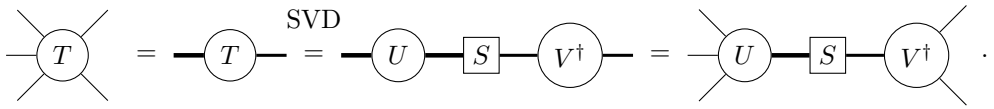
Essentially, $\mathbb{C}^{d_1 \times \ldots \times d_r} \cong \mathbb{C}^{\prod_{i=1}^{r} d_i}$, hence, the rank of a tensor is a fluid concept. This means, we can *group* indices $i_1, ..., i_n$ with dimensions $d_1, ..., d_n$ to a new index $\alpha = i_1 + d_1 i_2 + ... + (d_{n-1} i_n)$, and *split* an index i into several indices $(\alpha, \beta, ...)$. By doing so, we lower or raise the rank of the given tensors. In TNN, this corresponds to grouping or splitting legs, e.g.,



In this way, every contraction of higher rank can be written as matrix multiplication, which simplifies the computation of TNs due to the highly optimized matrix multiplication algorithms.

3.9.6 Singular Value Decomposition (SVD)

Given a matrix $T \in \mathbb{C}^{m \times n}$, we can use *singular value decomposition (SVD)*, to diagonalize T and write $T = USV^\dagger$ where $U \in \mathbb{C}^{m \times m}, V \in \mathbb{C}^{n \times n}$ are unitary and $S \in \mathbb{C}^{m \times n}$ is diagonal with non-negative, real entries. By grouping the indices, we can perform SVD on any rank-$m + n$ tensor. We then again split the indices, which, e.g., leads to



3.9.7 Tensor Network

We can now combine the above operations to define tensor networks (TNs) consisting of single tensors. The rank of the TN is given by the number of unconnected legs and its value by the sum over all internal indices of the product of the values of the single tensors that constitute the TN.

For bigger TNs, it is often unfeasible to resolve all contractions in one go. Luckily, the contractions can be performed one at a time in any order. The order in which we introduce and contract the tensors is known as *bubbling* and heavily influences the efficiency with which we can calculate the values of a TN.

3.9.8 Matrix Product States

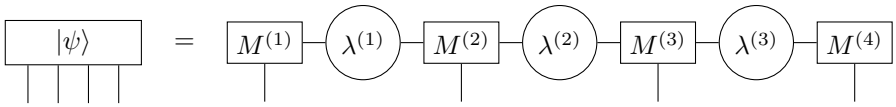
Now assume we are given a pure state on n qudits, i.e.,

$$|\psi\rangle = \sum_{i_1,\dots,i_n=1}^{d} c_{i_1\dots i_n} |i_1 \dots i_n\rangle .$$

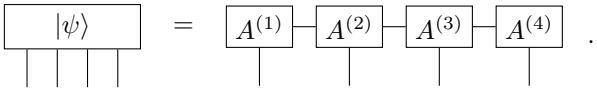
It's *Schmidt decomposition* is

$$|\psi\rangle = \sum_{i=1}^{d} c_i |f_i^{(1)}\rangle \otimes \dots \otimes |f_i^{(n)}\rangle ,$$

where the $\{f^{(k)}\}$ form a basis of the Hilbert space of the k^{th} qudit $\mathcal{H}_k$ [51]. We can then get this Schmidt decomposition by using SVD and grouping and splitting as explained in Section 3.9.6, e.g.,

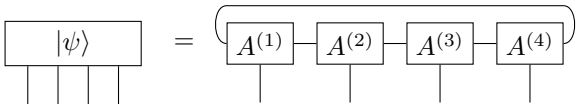


where the λ are diagonal matrices of singular values. Contracting the singular values into the local tensors M leads to the form



.

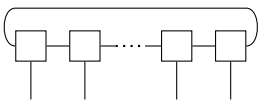
This kind of decomposition of a state is a so-called *matrix product state (MPS)*. Note that the name "matrix" product state can be misleading, as most involved tensors have a rank higher than two. Often, we instead use the convention[4]



or more generally,

$$|\psi[A^{(1)},\dots,A^{(n)}]\rangle = \sum_{i_1,\dots,i_n=1}^{d} \operatorname{tr}\left[A_{i_1}^{(1)} \dots A_{i_n}^{(n)}\right] |i_1 \dots i_n\rangle . \tag{3.9.1}$$

In the translationally invariant case, i.e., $A^{(1)} = \dots = A^{(n)} =: A$, this becomes

$|\psi[A]\rangle = \sum_{i_1,\dots,i_n=1}^{d} \operatorname{tr}\left[A_{i_1} \dots A_{i_n}\right] |i_1 \dots i_n\rangle =$  .

The uncontracted indices are referred to as *physical* indices, and the contracted indices are referred to as *virtual* or *bond* indices. The dimension of the bond indices is called the *bond dimension*. Writing states as MPS is only useful if the bond dimension is reasonably small. While most states cannot be written as an MPS with reasonably small bond dimension, many physical states can or can be approximated as such. One example is the GHZ state.

[4] This is an example of a (periodic) boundary condition.

Example 3.5 (GHZ state)**.** *By choosing A according to*

$$A_0 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

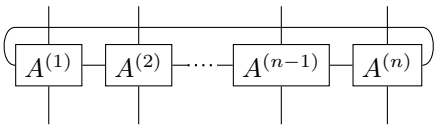
and normalizing, we get the Greenberger-Horne-Zeilinger (GHZ) state

$$|\psi_{GHZ}\rangle = \frac{1}{\sqrt{2}} \left(|0 \dots 0\rangle + |1 \dots 1\rangle \right).$$

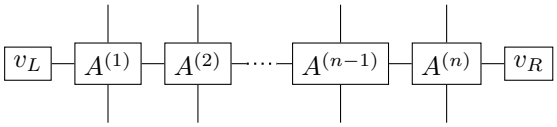
The mapping of a state to an MPS representation is usually ambiguous. This is known as *gauge freedom*. Two examples of gauge transformations are basis transformations on the virtual legs and blocking.

3.9.9 Matrix Product Operators

The equivalent of a MPS for operators, e.g., density operators, is a *matrix product operator (MPO)*. In the same way as before, we can write operators as



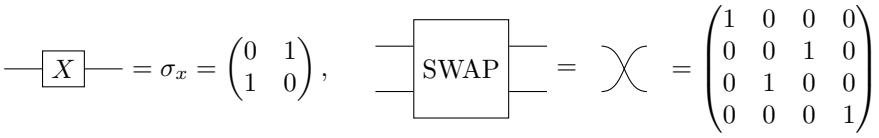
or



where we again want to keep the bond dimension as low as possible.

3.10 Quantum Computers

Quantum computers rely on quantum mechanical systems for computations. Based on logical gates, we name unitaries quantum gates. Examples of quantum gates on 1 or 2 qubits are



where the X gate performs a bit-flip on 1 qubit, and the SWAP gate swaps one qubit with another. Referring to classical computers being built from electrical circuits with wires and logical gates, quantum circuits are combinations of quantum state preparations, quantum gates, and measurements, which we have to add here due to the probabilistic nature of quantum mechanics. The way of depicting circuits is mostly the same as for classical circuits and tensor networks. For example, the circuit depicted in Figure 3.7 shows the preparation of two states ρ_A and ρ_B, the use of a unitary on their tensor product, and the subsequent tracing out of system A. The resulting state is then $\rho = \mathrm{tr}_A(U(\rho_A \otimes \rho_B)U^\dagger)$ [51].

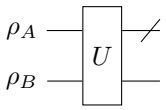


Figure 3.7: **Example Quantum Circuit.** First, the states ρ_A and ρ_B are prepared, then the unitary U is applied to their tensor product, and then system A is traced out.

Many current candidates for quantum computers focus on qubits as the fundamental units of information; other possibilities are, e.g., continuous-variable quantum computing [58] or quantum annealers [91]. Contenders for implementing qubits include trapped ion qubits [59, 183], superconducting qubits [184–188], photonic qubits, and neutral atom qubits [60–63, 189]. Current leading implementations have either a comparatively low number of physical qubits (up to 200) paired with high fidelities (more than 0.98, often higher, up to 0.997 for 1 qubit gates) and low noise [59, 63, 183, 185, 187–189], or a higher number of qubits (up to 1180) and unspecified fidelity and noise [61, 62, 184, 186, 190, 191]. Thus, with these current implementations, we are still in the era of *noisy intermediate-scale quantum (NISQ)* devices [192].

3.11 Quantum Machine Learning

The field of quantum machine learning (QML) is an emerging field combining the fields of ML and QI. One can generally divide it into the following sub-fields (see, e.g., [66–68]) by the type of data (first letter) and processing (second letter) used. The letter "Q" is then short for "quantum" and the letter "C" short for "classical".

CC ML encompasses methods using classical algorithms to process classical data, in the context of QML, with approaches derived from quantum information research, e.g., employing tensor networks to train NNs [66, 69].

QC ML uses classical algorithms to process quantum data for tasks in, e.g., quantum metrology [193], simulating quantum many-body systems [77], adaptive quantum computation [194], or quantum error correction (QEC) [41, 195–203]. A review of this field can be found in [204].

CQ ML uses quantum algorithms to process classical data mainly with the aim to speed up classical learning algorithms [65, 70–72]. This is by far the most explored subject under the term QML, so much so that other scenarios are not even mentioned in some reviews on QML [205]. One approach is to replace classical models in traditional ML algorithms with quantum ones [66, 206]. Although the literature on these methods outperforming classical ones is overwhelmingly positive, a metastudy found that when benchmarking influential quantum models [207–217] and out-of-the-box classical models on simple classical SL tasks, classical algorithms systematically outperform the current quantum ones. Additionally, the quantum algorithms usually perform better without entanglement [218]. Another potentially more promising field of study is creating CQ ML algorithms based on quantum subroutines with an already known speed-up [66]. An example of this is using Grover's algorithm for RL [64, 219]. An introduction to the field of CQ ML can be found in [220].

QQ ML uses quantum algorithms to process quantum data [221–230]. For quantum data as input of a SL algorithm, as opposed to only classical data, it has been shown that fully-

quantum protocols can be helpful [78–81], e.g., some problems with quantum states as input are learnable in polynomial time with quantum algorithms but not classical algorithms [231]. As for the other sub-fields, RL is much less explored than (un-)supervised learning and, in this case, much harder to define. Possible scenarios are, e.g., discussed in [232].

In this thesis, we want to focus on using either classical or quantum ML techniques for quantum data. As we already reviewed the classical ML methods needed, we will now discuss the QQ ML method we use in this thesis: dissipative quantum neural network (DQNN). Before, we will briefly describe other proposed quantum neural network (QNN) approaches.

3.11.1 QNN approaches

An overview of different QNN architectures is given in [233].

Most approaches of so-called QNNs fall into the field of CQ ML and are given by a combination of classical pre-processing, simple variational quantum circuits (i.e., quantum circuits depending on certain trainable parameters that are optimized classically) repeated in *layers* and classical post-processing [207, 208, 210–212, 218, 233–236]. The name is chosen in this way as classical NNs can generally be incorporated in QNNs [237]. In order for the QNNs to be expressive and not only able to approximate simple sine functions, one often has to make use of data re-uploading of classical data [209, 238]. The circuits can then be written as

$$U_{\mathrm{QNN}}(\vec{\theta}) = \prod_{l=L}^{1} S(\vec{x}) U^l(\vec{\theta}_l) W^l$$

where $S(\vec{x})$ is a data uploading scheme of a classical input $\vec{x}$, $U^l(\theta)$ are variational gates, and the W^l are fixed, typically entangling gates. Many different QNNs in the context of CQ ML use this ansatz [217, 218].

Fewer QNN approaches exist in the context of QQ ML, e.g., quantum generative adversarial networks (QGANs) [239, 240] or quantum convolutional neural networks (QCNNs) [241, 242] for many-body physics or QEC. In [243], the authors present a NN for cv QC able of approximating any unitary on N modes. Each layer of the QNN consists of a trainable parametrized Gaussian unitary, which leads to a linear transformation in phase space, and any fixed non-Gaussian gate on each of the modes, which leads to a non-linear transformation in phase space. Note that, without additional systems, this only leads to being able to approximate unitaries, not channels, on the N modes. Another example are DQNNs discussed next.

3.11.2 Dissipative Quantum Neural Networks

Dissipative quantum neural networks (DQNNs) were first introduced in [244] and further used and described in [104, 245–251]. If not mentioned otherwise, we will follow [244].

When dealing with quantum input and output data in the SL framework, we face training data of the form

$$S = \left(\left(\rho_1^{\mathrm{in}}, \sigma_1^{\mathrm{out}} \right), ..., \left(\rho_N^{\mathrm{in}}, \sigma_N^{\mathrm{out}} \right) \right) \in \left(\mathcal{D}(\mathcal{H}^{\mathrm{in}}) \times \mathcal{D}(\mathcal{H}^{\mathrm{out}}) \right)^{\times N}.$$

In the case of $\mathcal{H}^{\mathrm{in}} = \mathbb{C}^{d^{\mathrm{in}}}$, $\mathcal{H}^{\mathrm{out}} = \mathbb{C}^{d^{\mathrm{out}}}$ for some d^{in}, $d^{\mathrm{out}} \in \mathbb{N}$ for which there exist m^0, m^{L+1}, $d \in \mathbb{N}$ such that $d^{\mathrm{in}} = m^0 d$, $d^{\mathrm{out}} = m^{L+1} d$, we can employ DQNNs for SL.

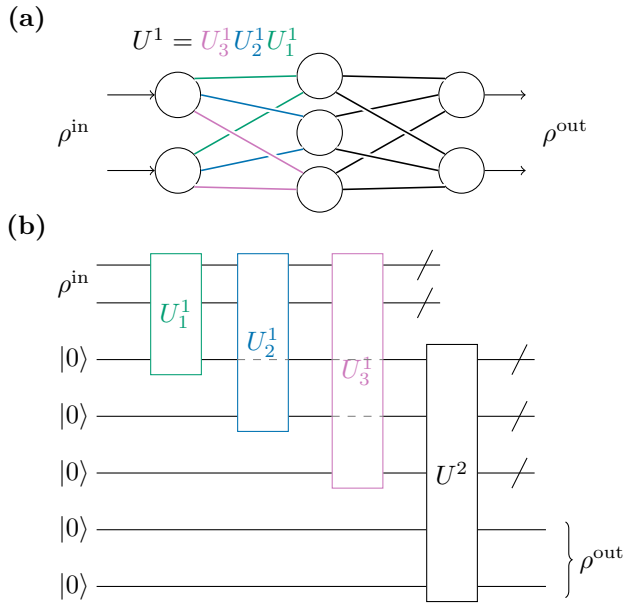


Figure 3.8: **DQNN architecture.** Panel (a) shows the graphical representation, and Panel (b) the corresponding quantum circuit of a fc ff DQNN with a two qudit input and output, and one hidden layer consisting of three neurons. With neuron j in layer l, we associate a qudit, and a unitary U_j^l that acts on qudit j in layer l and the connected neurons in layer $l-1$. Given an input ρ^{in}, the output of the DQNN is given by $\rho^{\text{out}} = \text{tr}_{0:L}\left(\mathcal{U}\left(\rho^{\text{in}} \otimes |0\ldots0\rangle_{1:L+1}\langle 0\ldots0|\right)\mathcal{U}^\dagger\right)$, where $U^{l+1} = U_{m_{l+1}}^{l+1}\ldots U_1^{l+1}$, $\mathcal{U} = U^{L+1}\ldots U^1$.

A DQNN is a quantum circuit organized in layers similar to an NN. We also use the same names to describe the different layers, i.e., *input, output,* and *hidden layers* as in the classical case. Each node represents a qudit, and with a connection between nodes of layer $l-1$ and node j of layer l, we associate a unitary U_j^l acting on the qudits which are represented by the connected nodes. In this way, only the connected neurons interact with each other. All unitaries associated with connections between layers $l-1$ and l are pooled together in a *layer unitary* $U^l = U_{m_l}^l \ldots U_1^l$ where m_l is the number of neurons in layer l. Here, and in the following, we do not write down the identities on other neurons and assume them to be absorbed in the unitaries implicitly. To transfer a state ρ^l from layer l to layer $l+1$, we tensor ρ^l to the ground state on the layer $l+1$ qudits $|0\ldots0\rangle^{l+1}\langle 0\ldots0|$, apply the layer unitary U^{l+1} to the composite state, and then trace out the qudits of layer l. Hence,

$$\rho^{l+1} := \mathcal{E}^{l+1}\left(\rho^l\right) := \text{tr}_l\left(U^{l+1}\left(\rho^l \otimes |0\ldots0\rangle^{l+1}\langle 0\ldots0|\right)U^{l+1\dagger}\right).$$

With L hidden layers and input ρ^{in}, the output of the DQNN becomes

$$\rho^{\text{out}} = \text{tr}_{0:L}\left(\mathcal{U}\left(\rho^{\text{in}} \otimes |0\ldots0\rangle_{1:L+1}\langle 0\ldots0|\right)\mathcal{U}^\dagger\right), \tag{3.11.1}$$

where $\mathcal{U} = U^{L+1}\ldots U^1$ is the *network unitary*, we write $i:j$ short for $i,\ldots,j$, and index the input layer by 0 and the output layer by $L+1$. The graphical and circuit representation of a DQNN and the notation are shown in Figure 3.8. DQNNs can be formalized with the following definition.

Definition and Proposition 3.8 (Fc ff dissipative quantum neural network (DQNN))**.** *Let $L \in \mathbb{N}$, $(m_l)_{l=0,\dots,L+1} \in \mathbb{N}^{L+2}$. An architecture m defines a function class, i.e., a set of functions, of* fc ff dissipative quantum neural networks (DQNNs) *given by*

$$\mathcal{F}_m^{DQNN} = \Bigg\{ \mathcal{N}_{\mathcal{U}} : \mathcal{T}(\mathbb{C}^{d^{m_0}}) \to \mathcal{T}(\mathbb{C}^{d^{m_{L+1}}}) \;\Big|\; \mathcal{N}_{\mathcal{U}}(\rho^0) = \rho^{L+1}, \\ \rho^l = \mathrm{tr}^{l-1}\left(U^l \left(\rho^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0| \right) U^{l\dagger} \right), \\ U^l = (\mathbb{1}^l_{1:m_l-1} \otimes U^l_{m_l}) \dots (U^l_1 \otimes \mathbb{1}^l_{2:m_l}), \\ U^l_j \in \mathcal{B}(\mathbb{C}^{d^{m_{l-1}+1}}),\ U^l_j U^{l\dagger}_j = \mathbb{1}, \\ \mathcal{U} = (\mathbb{1}^{0:L-1} \otimes U^{L+1}) \dots (U^1 \otimes \mathbb{1}^{2:L+1}) \Bigg\}$$

Given an input state $\rho^0 \in \mathcal{D}(\mathbb{C}^{d^{m_0}})$, and a DQNN $\mathcal{N}_{\mathcal{U}}$, its output is

$$\rho^{L+1} = \mathcal{N}_{\mathcal{U}}(\rho^0) = \mathrm{tr}_{0:L}\left(\mathcal{U} \left(\rho^0 \otimes |0\dots0\rangle_{1:L+1} \langle 0\dots0| \right) \mathcal{U}^\dagger \right) \in \mathcal{D}(\mathbb{C}^{d^{m_{L+1}}}).$$

We call the $\{U^l_j\}$ the perceptron unitaries*, the $\{U^l\}$ the* layer unitaries*, and $\mathcal{U}$ the network unitary. The* width *of the DQNN is $\|m\|_\infty$ and it's depth is L. We say the network is* shallow *if $L = 2$ and* deep *if $L > 2$.*

If we do not include additional measurements, DQNNs are not classical NNs in the narrower sense of the word as they do not use a non-linear activation function. However, due to Stinespring's theorem (see Proposition 3.1), they have the same property of being universal approximators, not for all classical functions, but quantum channels, which are the most general maps with (only) quantum states as input and output, i.e., the following proposition holds.

Proposition 3.3 (Universality of DQNNs [82])**.** *Every DQNN is a channel and for every channel $\mathcal{E} : \mathcal{T}(\mathbb{C}^{d^{\text{in}}}) \to \mathcal{T}(\mathbb{C}^{d^{\text{out}}})$ with $d^{\text{in}} = m^0 d$, $d^{\text{out}} = m^{L+1} d$, d, m^0, $m^{L+1} \in \mathbb{N}$ there exists a DQNN architecture $m \in \mathbb{N}^{L+2}$, where $L \in \mathbb{N}$, and network unitary $\mathcal{U}$ s.t. $\mathcal{E} = \mathcal{N}_{\mathcal{U}}$.*

Training Data and Cost Function

The training data for a DQNN is of the form

$$S = \left(\left(\rho_1^{\text{in}}, \sigma_1^{\text{out}} \right), \dots, \left(\rho_N^{\text{in}}, \sigma_N^{\text{out}} \right) \right) \in \left(\mathcal{D}(\mathcal{H}^{\text{in}}) \times \mathcal{D}(\mathcal{H}^{\text{out}}) \right)^{\times N}.$$

We write

$$\rho_x^{\text{out}} = \mathcal{N}_{\mathcal{U}}(\rho_x^{\text{in}})$$

for the output of the DQNN when inputting ρ_x^{in} and want this to be as close to the target output σ_x^{out} as possible.

If all target outputs σ_x^{out} are pure, i.e., we can write $\sigma_x^{\text{out}} = |\psi_x^{\text{out}}\rangle\langle\psi_x^{\text{out}}|$ $\forall x \in \{1, \dots, n\}$, the fidelity (see Equation (3.7.2)) is the essentially unique measure of closeness. We then use the infidelity as our loss

$$l(\mathcal{N}_{\mathcal{U}}, (\rho^{\text{in}}, \sigma^{\text{out}})) = 1 - F(\mathcal{N}_{\mathcal{U}}(\rho^{\text{in}}), \sigma^{\text{out}})), \tag{3.11.2}$$

and hence the cost function

$$C_S = 1 - \frac{1}{N}\sum_{x=1}^{N} F(\rho_x^{\text{out}}, \sigma_x^{\text{out}}) = 1 - \frac{1}{N}\sum_{x=1}^{N} \text{tr}(\rho_x^{\text{out}}\sigma_x^{\text{out}}). \tag{3.11.3}$$

If not all target outputs are pure, the fidelity becomes hard to handle in the derivation of the training algorithm. Thus, we then use the squared Hilbert-Schmidt distance (or Schatten-2-distance, see Equation (3.7.1)) as our loss

$$l(\mathcal{N}_{\mathcal{U}}, (\rho^{\text{in}}, \sigma^{\text{out}})) = \|\mathcal{N}_{\mathcal{U}}(\rho^{\text{in}}) - \sigma^{\text{out}})\|_2^2, \tag{3.11.4}$$

and hence

$$C_S = \frac{1}{N}\sum_{x=1}^{N} \text{tr}((\rho_x^{\text{out}} - \sigma_x^{\text{out}})^2) \tag{3.11.5}$$

as the cost instead [104, 252].

Classical Training

First, let us look at a way to train DQNNs classically. We can either see this as a simulation of training DQNNs on a general quantum computer or as a training algorithm for classically training DQNNs on classical representations of quantum states, which is further discussed in [253]. During training, we update each unitary U_j^l by multiplying it with a matrix $\exp P_j^l = \exp i\epsilon K_j^l$. By choosing Hermitian matrices K_j^l, we make sure that the unitaries U_j^l stay unitary. By minimizing the difference after and before the update for $\epsilon \to 0$, we get Algorithm 9. The derivation can be found in Appendix A.1 and [244, 245] for pure states and in Appendix A.2 and in [252] for mixed states.

Similar to classical NNs, the optimization algorithm can be split into a feed-forward part, where the input is layer-by-layer propagated forward through the DQNN, and a backpropagation part, where the target output or error is layer-by-layer propagated backward through the DQNN. The update matrices are then given by a trace over a commutator between those parts, each multiplied by a part of the unitaries in that layer. We used the notation that $U_{j_1:j_2}^l = U_{j_2}^l \dots U_{j_1}^l$ for $j_1 \leq j_2$, and $U_{j_1:j_2}^l = \mathbb{1}$ for $j_1 > j_2$. Numerical results can be found in [244, 245] for pure states and in [252] for mixed states.

NISQ Devices

On NISQ devices, we can only use certain parametrized unitaries and build general unitaries out of those. A DQNN is then written by parametrized unitaries with a total of N_p parameters. The parameters are stored in a vector $\vec{p_t} \in \mathbb{R}^{N_p}$ that is initialized and then updated using classical gradient descent, i.e., $\vec{p_{t+1}} = \vec{p_t} + \vec{dp}$ where $\vec{dp} = \eta \nabla C\,(\vec{p_t})$. We estimate the gradient using

$$(\nabla C\,(\vec{p_t}))_k = \frac{C\,(\vec{p_t} + \epsilon\vec{e_k}) - C\,(\vec{p_t} - \epsilon\vec{e_k})}{2\epsilon}$$

with $(\vec{e_k})^j = \delta_k^j,\ k, j = 1, ..., N_p$ and the cost function by using the SWAP method (further described in [245]) for pure outputs [250].

Algorithm 9 Classical Training Algorithm of ff DQNNs

Input: $((\rho_x^{\text{in}}, \sigma_x^{\text{out}}))_{x=1,\dots,N}$, $(m_l)_{l=0,\dots,L+1}$ or $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U_j^l)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits* ◁

      $U_j^l \leftarrow$RandomUnitaryHaar($m_{l-1}+1$ qudits)

      $U_j^l \leftarrow \mathbb{1}_{1:j-1}^l \otimes U_j^l \otimes \mathbb{1}_{j+1:m_l}^l$

for t=1,...,T **do**

  for x=1,...,N **do** ▷ *Feed-forward*

    $\rho_x^0 \leftarrow \rho_x^{\text{in}}$

    for l=1,...,L+1 **do**

      $\rho_x^l \leftarrow \text{tr}^{l-1}\left(U^l \left(\rho_x^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0|\right) U^{l\dagger}\right)$

  for x=1,...,N **do** ▷ *Backpropagation*

    if $\forall x \in \{1:n\}\ \exists\, \psi_x^{\text{out}} \in \mathcal{H}^{\text{in}}$ such that $\sigma_x^{\text{out}} = |\psi_x^{\text{out}}\rangle \langle\psi_x^{\text{out}}|$ **then**

      $\sigma_x^{L+1} \leftarrow |\psi_x^{\text{out}}\rangle \langle\psi_x^{\text{out}}|$

    else

      $\sigma_x^{L+1} \leftarrow \sigma_x^{\text{out}} - \rho_x^{L+1}$

    for l=0,...,L **do**

      $\sigma_x^l \leftarrow \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right) U^{l+1\dagger} \left(\mathbb{1}^l \otimes \sigma_x^{l+1}\right) U^{l+1}\right)$

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      $$M_j^l \leftarrow \frac{1}{N}\sum_{x=1}^{N}\Big[U_{1:j}^l\Big(\rho_x^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0|\Big)U_{1:j}^{l\dagger}, U_{j+1:m_l}^{l\dagger}\Big(\mathbb{1}^{l-1} \otimes \sigma_x^l\Big)U_{j+1:m_l}^l\Big]$$

      $P_j^l \leftarrow -2^{m_{l-1}}\eta\, \text{tr}_{1:j-1}^l\Big(\text{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big)$

      $U_j^l \leftarrow \exp\left(P_j^l\right) U_j^l$

  ▷ *Calculate Cost* ◁

  if $\forall x \in \{1:n\}\ \exists\, \psi_x^{\text{out}} \in \mathcal{H}^{\text{in}}$ such that $\sigma_x^{\text{out}} = |\psi_x^{\text{out}}\rangle \langle\psi_x^{\text{out}}|$ **then**

    $C_t \leftarrow 1 - \frac{1}{N}\sum_{x=1}^{N} \text{tr}\left(\rho_x \sigma_x\right)$

  else

    $C_t \leftarrow \frac{1}{N}\sum_{x=1}^{N} \text{tr}\left(\left(\rho_x^{L+1} - \sigma_x^{\text{out}}\right)^2\right)$

3.11.3 Challenges

While the main reason for looking into quantum algorithms in ML is the potential for a quantum advantage, especially when analyzing data from quantum mechanical processes, there are also challenges. As an example, we discuss two of them; others include the search for QML architectures and the noise in current quantum systems. This section is adapted from [254].

Barren plateaus. Like in classical ML, algorithms making use of quantum information often have to deal with local minima in the loss function. However, this is not the only challenge we face regarding the loss landscape. We also encounter *Barren plateaus*, i.e., the loss landscape grows exponentially flatter, and the valley of the global optimum shrinks exponentially with growing system size [255, 256] (strategies for dealing with this are, e.g., explained in [257]). This phenomenon appears in many QNNs [255, 258, 259], including DQNNs [248]. It arises due to the hypothesis space being too expressive [258, 260] (strategies for dealing with this are explored in [261–263]), global observables [264, 265], noise [266], or too much entanglement [248, 267] (strategies for dealing with this are explored in [268]).

Data sets. As for any ML algorithm, we need datasets for benchmarking in QML. In CQ ML, embedding the classical data in quantum states is still an active field of research [269]. Although some proposals exist [213, 217, 270], it has been shown that embeddings that are both hard to simulate classically and practically useful can still lead to further Barren plateaus [271]. In QQ ML, only a few data sets (e.g. [272, 273]) exist.

4

Dissipative Quantum Recurrent Neural Networks

The just considered ff DQNNs serve as a powerful technique for analyzing independent and identically distributed (i.i.d.) quantum data. However, in quantum mechanics, we are faced with a number of tasks where the data is not i.i.d. but sequential. Examples include the time-evolution of states w.r.t. a time-dependent Hamiltonian, applications in quantum control [274, 275], or the examination of quantum channels with memory [90]. As we discussed in Section 2.1.4, RNNs are a key resource for processing sequential data. Hence, we want to explore a recurrent version of DQNNs, called dissipative quantum recurrent neural network (DQRNN), which we will introduce in Section 4.1. Since we use DQNNs as a base, they are universal for analyzing sequential quantum data in the sense that they can approximate any causal quantum automaton. We will further discuss this in Section 4.2.

Especially in the context of CQ ML, there exists a number of different so-called quantum recurrent neural network (QRNN) architectures [276–279]. In the context of QQ ML, this is less so, and only unitary QRNNs in the cv context are discussed [243]. These approaches are reviewed in Section 4.3.

Section 4.4 focuses on the different kinds of training data and cost functions we will discuss in this chapter. As we are still in the NISQ era, we simulate our results in this chapter on a classical computer. Hence, at the heart of this chapter, we develop classical training algorithms in Section 4.5 for the presented data sets and cost functions. These algorithms can be seen either as a simulation of employing DQRNNs on a QC or as a classical learning algorithm if we are given or obtain a classical representation of the training data. We then use these algorithms to analyze the delay channel, a standard example for a quantum channel with memory, and the time evolution of states under a time-dependent Hamiltonian, as time predictions are a standard use case for classical RNNs. These numerical results are presented in Section 4.6. In Section 4.7, we present a possible training algorithm on NISQ devices.

Most of this chapter, except for the case of entangled training data, is discussed in the work [104].

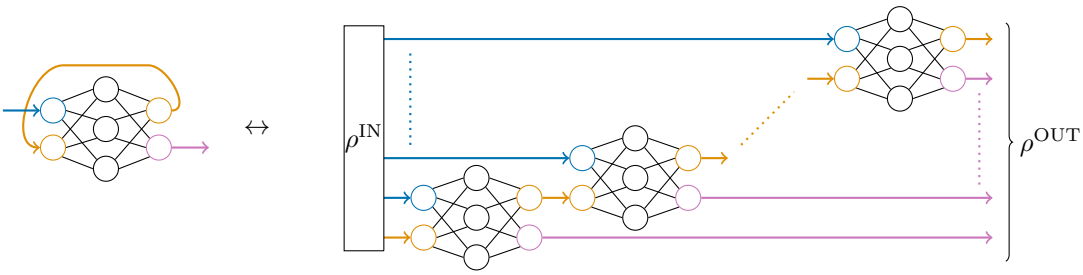


Figure 4.1: **Dissipative quantum recurrent neural network (DQRNN) Architecture.** We define a DQRNN as an iteration over the memory system (orange) of a ff DQNN with L hidden layers (here $L = 1$) where the 0^{th} layer is split into input neurons (blue, here 1) and memory neurons (here 1) and the $L+1^{\mathrm{th}}$ layer is split into output neurons (purple, here 1) and memory neurons (here 1). For a concrete number of iterations N, we can input any state ρ^{IN} on $\mathcal{H}^{\mathrm{mem}} \otimes \mathcal{H}_1^{\mathrm{in}} \otimes \cdots \otimes \mathcal{H}_N^{\mathrm{in}}$ and get out a state ρ^{OUT} on $\mathcal{H}_1^{\mathrm{out}} \otimes \cdots \otimes \mathcal{H}_N^{\mathrm{out}} \otimes \mathcal{H}^{\mathrm{mem}}$.

4.1 Architecture

In classical RNNs, the output of some layers of a NN is used again as input in the same or previous layers. The same idea can be utilized with DQNNs. As DQNNs are linear, we can assume, w.l.o.g., that part of the output of a DQNN is used again as part of the input. The reused part is called the *memory*, referring to quantum channels with memory and classical RNNs. In particular, we split both the Hilbert space of the 0^{th} layer (total input) $\mathcal{H}^{\mathrm{intotal}} = \mathcal{H}^0$ and $(L+1)^{\mathrm{th}}$ layer (total output) $\mathcal{H}^{\mathrm{outotal}} = \mathcal{H}^{L+1}$ into two Hilbert spaces each, i.e., we write $\mathcal{H}^0 = \mathcal{H}^{\mathrm{mem}} \otimes \mathcal{H}^{\mathrm{in}}$ and $\mathcal{H}^{L+1} = \mathcal{H}^{\mathrm{out}} \otimes \mathcal{H}^{\mathrm{mem}}$. A part of the neurons in the last layer (neurons in $\mathcal{H}^{\mathrm{mem}}$) is reused as input for a part of the neurons in the 0^{th} layer (again, neurons in $\mathcal{H}^{\mathrm{mem}}$). This, in total, defines a *dissipative quantum recurrent neural network (DQRNN)*. If we assume that all neurons are associated with qudits on the same d-dimensional Hilbert space, the number of memory qudits in the 0^{th} and $(L+1)^{\mathrm{th}}$ layer has to be the same. This is due to us not employing a channel between memory output and memory input and to the impossibility of copying or cloning quantum states [280, 281].

Like for classical RNNs, discussed in Section 2.1.4, and quantum channels with memory, discussed in Section 3.6, we can depict a DQRNN either in a compressed or an unfolded version. These are both shown in Figure 4.1. The compressed version shows a DQNN in which the output of some of the neurons in the last layer is used as input of the same number of neurons in the 0^{th} layer. In the figure, the output of the first neuron in the second layer is used as input of the second neuron in layer zero. Figure 4.1 can, in particular, be understood as a special case of Figure 3.6, showing the compressed and unfolded version of a quantum channel with memory. The general channel in Figure 3.6 is replaced by a DQNN in Figure 4.1. For N iterations of the underlying DQNN $\mathcal{N}_{\mathcal{U}}$, the DQRNN can thus, in the same way as a quantum channel with memory for N iterations of the underlying quantum channel, be written as

$$\mathcal{M}_{N,\mathcal{U}} = \left(\mathbb{1}_{1,\ldots,N-1}^{\mathrm{out}} \otimes \mathcal{N}_{\mathcal{U}}\right) \circ \cdots \circ \left(\mathbb{1}_1^{\mathrm{out}} \otimes \mathcal{N}_{\mathcal{U}} \otimes \mathbb{1}_{3,..,N}^{\mathrm{in}}\right) \circ \left(\mathcal{N}_{\mathcal{U}} \otimes \mathbb{1}_{2,..,N}^{\mathrm{in}}\right).$$

The input can then be any state ρ^{IN} on $\mathcal{H}_N^{\mathrm{IN}} := \mathcal{H}_0^{\mathrm{mem}} \otimes \mathcal{H}_1^{\mathrm{in}} \otimes \cdots \otimes \mathcal{H}_N^{\mathrm{in}}$, and the corresponding output is given by

$$\rho^{\mathrm{OUT}} = \mathcal{M}_{N,\mathcal{U}}\left(\rho^{\mathrm{IN}}\right)$$

which is a state on $\mathcal{H}_N^{\text{OUT}} := \mathcal{H}_1^{\text{out}} \otimes \cdots \otimes \mathcal{H}_N^{\text{out}} \otimes \mathcal{H}_N^{\text{mem}}$. DQRNNs are formalized in the following definition.

Definition 4.1 (Fc dissipative quantum recurrent neural network (DQRNN))**.** *Let $\mathcal{H}^{\text{in}}$, $\mathcal{H}^{\text{out}}$, and $\mathcal{H}^{\text{mem}}$ be Hilbert spaces of dimension $m^{\text{in}}d$, $m^{\text{out}}d$, and $m^{\text{mem}}d$ where d, m^{in}, m^{out}, $m^{\text{mem}} \in \mathbb{N}$. Let $L \in \mathbb{N}$, $(m_l)_{l=0,\dots,L+1} \in \mathbb{N}^{L+2}$, where $m_0 = m^{\text{mem}} + m^{\text{in}}$ and $m_{L+1} = m^{\text{mem}} + m^{\text{out}}$. A fc dissipative quantum recurrent neural network (DQRNN) is represented by an* underlying *DQNN $\mathcal{N}_{\mathcal{U}} : \mathcal{T}(\mathcal{H}^{\text{mem}}) \otimes \mathcal{T}(\mathcal{H}^{\text{in}}) \to \mathcal{T}(\mathcal{H}^{\text{out}}) \otimes \mathcal{T}(\mathcal{H}^{\text{mem}})$ with architecture m. The tuple (m^{mem}, m) defines the* architecture *of the DQRNN. This architecture (m^{mem}, m) and an iteration number $N \in \mathbb{N}$ define a function class, i.e., a set of functions,*

$$\mathcal{F}^{DQRNN}_{(m^{\text{mem}},m),N} = \left\{ \mathcal{M}_{N,\mathcal{U}} : \mathcal{T}(\mathcal{H}^{\text{mem}}) \otimes \mathcal{T}(\mathcal{H}^{\text{in}})^{\otimes N} \to \mathcal{T}(\mathcal{H}^{\text{out}})^{\otimes N} \otimes \mathcal{T}(\mathcal{H}^{\text{mem}}) \,\middle|\, \mathcal{N}_{\mathcal{U}} \in \mathcal{F}^{DQNN}_{m}, \right.$$
$$\left. \mathcal{M}_{N,\mathcal{U}} = \left(\mathbb{1}^{\text{out}}_{1,\dots,N-1} \otimes \mathcal{N}_{\mathcal{U}}\right) \circ \cdots \circ \left(\mathbb{1}^{\text{out}}_{1} \otimes \mathcal{N}_{\mathcal{U}} \otimes \mathbb{1}^{\text{in}}_{3,..,N}\right) \circ \left(\mathcal{N}_{\mathcal{U}} \otimes \mathbb{1}^{\text{in}}_{2,..,N}\right) \right\}$$

in which we iterate over the underlying DQNN N times.

In the same way as classical RNNs can be seen as very deep, sparse ff NNs sharing a lot of their parameters in their unfolded versions, we can see unfolded DQRNNs as deep, sparse ff DQNNs sharing a lot of unitaries.

In addition to indexing the Hilbert spaces, states, unitaries, and channels by layer, we also use the number of iterations of the underlying DQNN as an index when handling DQRNNs, we write, e.g.,

$$\mathcal{N}_{\mathcal{U}_x} = \mathbb{1}^{\text{out}}_{1,\dots,x-1} \otimes \mathcal{N}_{\mathcal{U}} \otimes \mathbb{1}^{\text{in}}_{x+1,..,N}$$

for the x^{th} iteration, where $x = 1, ..., N$. Hence, we can write

$$\mathcal{M}_{N,\mathcal{U}} = \mathcal{N}_{\mathcal{U}_N} \circ \cdots \circ \mathcal{N}_{\mathcal{U}_1}$$

and

$$\rho^{\text{OUT}} = \mathcal{M}_{N,\mathcal{U}}\left(\rho^{\text{IN}}\right) = \left(\mathcal{N}_{\mathcal{U}_N} \circ \cdots \circ \mathcal{N}_{\mathcal{U}_1}\right)\left(\rho^{\text{IN}}\right). \tag{4.1.1}$$

4.2 Universality

DQRNNs can be used to describe any quantum channel with memory with finite input, output, and memory spaces.

Lemma 4.1. *Every DQRNN is a channel with memory, and for every channel with memory $\mathcal{E} : \mathcal{T}(\mathbb{C}^{d^{\text{mem}}}) \otimes \mathcal{T}(\mathbb{C}^{d^{\text{in}}}) \to \mathcal{T}(\mathbb{C}^{d^{\text{out}}}) \otimes \mathcal{T}(\mathbb{C}^{d^{\text{mem}}})$ with $d^{\text{in}} = m^0 d$, $d^{\text{out}} = m^{L+1}d$, $d^{\text{mem}} = m^{\text{mem}}d$, d, m^0, $m^{L+1} \in \mathbb{N}$ there exists a DQRNN with architecture (m, m^{mem}) and an underlying DQNN $\mathcal{N}_{\mathcal{U}}$ with network unitary $\mathcal{U}$, and architecture $m \in \mathbb{N}^{L+2}$, where $L \in \mathbb{N}$, s.t. $\mathcal{E} = \mathcal{N}_{\mathcal{U}}$.*

Proof. The strategy for building DQRNNs from DQNNs described above is the same as the one for concatenated memory channels as described in Definition 3.5. As every DQNN is a channel, and every channel with finite in- and output can be written as a DQNN, the Lemma follows by construction. □

This means that DQRNNs, like quantum channels with memory [90], together with initialization, can be used to represent any causal quantum automaton with finite input and output spaces. Hence, DQRNNs are universal approximators for causal quantum automata.

4.3 Related Approaches

Many very different *quantum recurrent neural network (QRNN)* architectures were already presented, most of them being hybrid quantum-classical models [276–279] missing a fully quantum RNN for qudits. We now summarize a widely used approach (mainly applied to *stochastic filtering*, i.e., extracting a signal from a noisy signal) and the ones sharing similarities with our approach (used primarily for time series prediction).

A popular approach first presented in [282] under a different name uses a classical input fed into a classical NN. The function defined by that NN then gives rise to a potential and, hence, to a (nonlinear) Schrödinger equation solved by a wave function. By determining the position's expectation value, a classical output is generated that is also subtracted from the following input. Thereby, the recurrency is generated. This ansatz is mainly used for stochastic filtering [283,284], e.g., of EEG [285] or EMG [286] signals, for describing target tracking through eye movement [286] or in continuous-variable quantum key distribution [287].

Other approaches like the ones from [288] or [289] include a kind of quantum neural network (QNN) instead of classical NNs in LSTM cells, GRUs or encoder-decoders. Thereby, the QNN often consists of assigning a quantum state to a classical input, applying a variational quantum circuit, making a measurement on the result or determining the expectation value of some operator, and sometimes using some activation function on it. The variational quantum circuit, then, is the subject of the optimization.

Similar to that ansatz and ours is the approach to encode classical data in an input state on qubits, apply a (highly-structured) parametrized quantum circuit, and measure a portion of the qubits. The remaining qubits are left untouched, and the next state coming from the classical input is tensored to them, which generates the recurrence. As before, the variational quantum circuit is optimized in the training process. This ansatz is then used for classical data [290–292].

The only fully quantum RNN was presented in [243] for a system of continuous variables that also can be used to encode classical data in quantum states. To construct QNNs, Gaussian gates are used instead of the affine transformations in classical NNs and a specific but arbitrary non-Gaussian gate is used instead of the activation function. In the recurrent case, a part of the output is again used as part of the following input. Example calculations were only provided for the hybrid quantum-classical feedforward case. Here, only the Gaussian gates are optimized.

As already explained, we use the ansatz from [82] for QNNs and introduce the recurrence in the same way as in [243] to construct fully quantum RNNs. While the underlying QNN in [243] can only approximate unitaries, DQNNs have the capability to approximate more general channels. This allows us to learn general quantum channels with memory.

Another approach sharing similarities with several others to QRNNs is the one of *quantum reservoir computing (QRC)*. Here, a classical input is encoded in a quantum state on system

A and tensored to a state on system B (the reservoir). This state then evolves w.r.t. an (up to some point) random Hamiltonian before a partial measurement is performed. The result is then used as input for a linear map, which yields the output. The major difference between QRC and most of the similar QRNN approaches is that in QRC, the linear map yielding the output is optimized. In contrast, in the QRNN approaches some kind of quantum circuit – corresponding to the Hamiltonian here – and not the output function is optimized [293–296].

4.4 Training Data and Cost

As already discussed, for a DQRNN with N iterations over the underlying DQNN, the input has to be in $\mathcal{B}(\mathcal{H}_N^{\text{IN}})$, and the output is in $\mathcal{H}_N^{\text{OUT}}$, where

$$\mathcal{H}_N^{\text{IN}} = \mathcal{H}^{\text{mem}} \otimes \mathcal{H}^{\text{in}^{\otimes N}}, \quad \mathcal{H}_N^{\text{OUT}} = \mathcal{H}^{\text{out}^{\otimes N}} \otimes \mathcal{H}^{\text{mem}}.$$

We also call $\mathcal{H}^{\text{in}}$, $\mathcal{H}^{\text{mem}}$ and $\mathcal{H}^{\text{out}}$ the *local* input, memory and output Hilbert spaces, respectively, and $\mathcal{H}_N^{\text{IN}}$ and $\mathcal{H}_N^{\text{OUT}}$ the *global* input and output Hilbert spaces, respectively. Each of our training pairs should be a global input-output pair consisting of quantum states on those same global input and output Hilbert spaces, i.e.,

$$S_\alpha = \left(\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}}\right) \in \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{IN}}) \times \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{OUT}}).$$

As we want learn the unitaries of the underlying DQNN, not all of the training pairs have to have the same N_α. Thus, most generally, the training set consists of M training pairs $S_\alpha \in \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{IN}}) \times \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{OUT}})$, where $\alpha = 1, ..., M$. Hence, the training set is of the form

$$S = \{S_\alpha\}_{\alpha=1}^M = \left\{\left(\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}}\right)\right\}_{\alpha=1}^M \in \bigtimes_{\alpha=1}^M \left(\mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{IN}}) \times \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{OUT}})\right) \tag{4.4.1}$$

where M is the number of *runs* of the DQRNN and N_α is the number of iterations over the underlying DQNN in run α.

Like for the DQNNs (see Section 3.11.2, Equations (3.11.2) and (3.11.4)), we use the infidelity as loss function l if the target states are pure and the Hilbert-Schmidt distance if they are not. We can compare the network output to the target output either globally (on $\mathcal{H}^{\text{OUT}}$) or locally (on $\mathcal{H}^{\text{out}}$).

Globally comparing

$$\rho_\alpha^{\text{OUT}} = \mathcal{M}_{N_\alpha,\mathcal{U}}\left(\rho_\alpha^{\text{IN}}\right)$$

to $\sigma_\alpha^{\text{OUT}}$ leaves us with the loss

$$\begin{aligned} l_{\text{global, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) &= 1 - F(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}}), \sigma^{\text{OUT}})) \\ &= 1 - \text{tr}\left(\mathcal{M}_{N_\alpha,\mathcal{U}}\left(\rho_\alpha^{\text{IN}}\right)\sigma_\alpha^{\text{OUT}}\right), \end{aligned}$$

for pure target outputs $\sigma^{\text{OUT}} = |\psi^{\text{OUT}}\rangle\langle\psi^{\text{OUT}}|$ for $\psi^{\text{OUT}} \in \mathcal{H}_N^{\text{out}}$, and

$$\begin{aligned} l_{\text{global, mixed}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) &= \|\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}}) - \sigma^{\text{OUT}})\|_2^2 \\ &= \text{tr}\left(\sqrt{\left(\mathcal{M}_{N,\mathcal{U}}\left(\rho^{\text{IN}}\right) - \sigma^{\text{OUT}}\right)^\dagger \left(\mathcal{M}_{N,\mathcal{U}}\left(\rho^{\text{IN}}\right) - \sigma^{\text{OUT}}\right)}^2\right), \end{aligned}$$

for mixed outputs, where we used the definition of the fidelity (Equation (3.7.2)) and Hilbert-Schmidt norm (Equation (3.7.1)) in the last steps.

According to Equation (2.1.2), the cost C is then given by

$$\begin{aligned} C_{\text{global, pure},S} &= \frac{1}{M}\sum_{\alpha=1}^{M} l_{\text{global, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}})) \\ &= 1 - \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}(\rho_\alpha^{\text{OUT}}\sigma_\alpha^{\text{OUT}}). \end{aligned}$$

for pure target outputs $\sigma_\alpha^{\text{OUT}} = |\psi_\alpha^{\text{OUT}}\rangle\langle\psi_\alpha^{\text{OUT}}|$ for $\psi_\alpha^{\text{OUT}} \in \mathcal{H}_{N_\alpha}^{\text{OUT}}$, and

$$\begin{aligned} C_{\text{global, mixed},S} &= \frac{1}{M}\sum_{\alpha=1}^{M} l_{\text{global, mixed}}(\mathcal{M}_{N,\mathcal{U}}, (\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}})) \\ &= \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}\left(\sqrt{(\rho_\alpha^{\text{OUT}} - \sigma_\alpha^{\text{OUT}})^\dagger (\rho_\alpha^{\text{OUT}} - \sigma_\alpha^{\text{OUT}})}^2\right) \\ &= \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}((\rho_\alpha^{\text{OUT}} - \sigma_\alpha^{\text{OUT}})^2) \end{aligned}$$

for mixed targets as both ρ_α^{OUT} and $\sigma_\alpha^{\text{OUT}}$ are quantum states, which are by definition Hermitian, and hence $(\rho_\alpha^{\text{OUT}} - \sigma_\alpha^{\text{OUT}})^\dagger = (\rho_\alpha^{\text{OUT}} - \sigma_\alpha^{\text{OUT}})$.

If the states ρ_α^{IN} and $\sigma_\alpha^{\text{OUT}}$ are product states, i.e., $\rho_\alpha^{\text{IN}} = \rho_{0\,\alpha}^{\text{mem}} \otimes \rho_{1\,\alpha}^{\text{in}} \otimes \cdots \otimes \rho_{N_\alpha\,\alpha}^{\text{in}}$ and $\sigma_\alpha^{\text{OUT}} = \sigma_{1\,\alpha}^{\text{out}} \otimes \cdots \otimes \sigma_{N_\alpha\,\alpha}^{\text{out}} \otimes \sigma_{N_\alpha\,\alpha}^{\text{mem}}$, we can write

$$S = \{S_\alpha\}_{\alpha=1}^{M} = \left\{\left(\rho_{0\,\alpha}^{\text{mem}}, \left(\left(\rho_{x\,\alpha}^{\text{in}}, \sigma_{x\,\alpha}^{\text{out}}\right)\right)_{x=1}^{N_\alpha}, \sigma_{N_\alpha\,\alpha}^{\text{mem}}\right)\right\}_{\alpha=1}^{M}$$

instead of Equation (4.4.1). Then, it can make sense to compare the separate outputs locally, i.e., comparing $\sigma_{x\,\alpha}^{\text{out}}$ to

$$\rho_{x\,\alpha}^{\text{out}} = \bar{\text{tr}}_x^{\text{out}}\left(\mathcal{M}_{\mathcal{U}}(\rho_\alpha^{\text{IN}})\right),$$

where $\bar{\text{tr}}$ traces out everything but the indicated layer. For this task, we use the local loss

$$l_{\text{local, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) = 1 - \frac{1}{N}\sum_{x=1}^{N} F(\bar{\text{tr}}_x^{\text{out}}\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}})\right), \sigma_x^{\text{out}})), \tag{4.4.2}$$

and cost

$$C_{\text{local, pure},S} = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha} \text{tr}(\rho_{x\,\alpha}^{\text{out}}\sigma_{x\,\alpha}^{\text{out}}). \tag{4.4.3}$$

for pure target outputs $\sigma_{x\,\alpha}^{\text{out}} = |\psi_{x\,\alpha}^{\text{out}}\rangle\langle\psi_{x\,\alpha}^{\text{out}}|$ for $\psi_{x\,\alpha}^{\text{out}} \in \mathcal{H}^{\text{out}}$ and the local loss

$$l_{\text{local, mixed}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) = 1 - \frac{1}{N}\sum_{x=1}^{N} \|\bar{\text{tr}}_x^{\text{out}}\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}})\right) - \sigma_x^{\text{OUT}})\|_2^2, \tag{4.4.4}$$

and cost

$$C_{\text{local, mixed},S} = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha} \text{tr}((\rho_{x\,\alpha}^{\text{out}} - \sigma_{x\,\alpha}^{\text{out}})^2) \tag{4.4.5}$$

for mixed targets.

Often, we will leave out $\sigma^{\text{mem}}_{N_\alpha\,\alpha}$ and/or $\rho^{\text{mem}}_{0\,\alpha}$ when they are unknown. In that case, we will set $\sigma^{\text{mem}}_{N_\alpha\,\alpha} = \mathbb{1}^{\text{mem}}_{N_\alpha\,\alpha}$ and $\rho^{\text{mem}}_{0\,\alpha} = |0\dots0\rangle^{\text{mem}}_{0\,\alpha}\langle 0\dots0|$ as these are the terms that would appear if we calculate the cost or its derivatives in the case of unknown memory states.

Note that, for pure states, the local cost is always smaller or equal to the global cost, as shown in the following Lemma.

Lemma 4.2 (Comparison of the local and global cost [104])**.** *Let $\mathcal{H}^{\text{in}}$, $\mathcal{H}^{\text{out}}$, $\mathcal{H}^{\text{mem}}$ be finite dimensional Hilbert spaces, $N \in \mathbb{N}$, $\mathcal{M}_{N,\mathcal{U}}$ a DQRNN, $\rho^{\text{IN}} \in \mathcal{D}\left(\mathcal{H}^{\text{mem}} \otimes (\mathcal{H}^{\text{in}})^{\otimes N}\right)$, and $\sigma^{\text{OUT}} \in \mathcal{D}\left((\mathcal{H}^{\text{out}})^{\otimes N} \otimes \mathcal{H}^{\text{mem}}\right)$. It then is*

$$l_{\textit{local, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) \leq l_{\textit{global, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})).$$

Proof. The fidelity fulfills the *data-processing inequality* [297], i.e., for two Hilbert space $\mathcal{H}_1, \mathcal{H}_2$, two states τ, ξ on $\mathcal{H}_1$ and a channel $\mathcal{E} : \mathcal{T}(\mathcal{H}_1) \to \mathcal{T}(\mathcal{H}_2)$, it is

$$F(\mathcal{E}(\tau), \mathcal{E}(\xi)) \geq F(\tau, \xi).$$

The partial trace is a channel and $\sigma^{\text{out}}_x = \bar{\text{tr}}^{\text{out}}_x\left(\sigma^{\text{OUT}}_x\right)$, hence

$$\begin{aligned}
l_{\text{local, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})) &= 1 - \frac{1}{N}\sum_{x=1}^{N} F\left(\bar{\text{tr}}^{\text{out}}_x\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}})\right), \sigma^{\text{out}}_x\right) \\
&= 1 - \frac{1}{N}\sum_{x=1}^{N} F\left(\bar{\text{tr}}^{\text{out}}_x\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}})\right), \bar{\text{tr}}^{\text{out}}_x\left(\sigma^{\text{OUT}}\right)\right) \\
&\leq 1 - \frac{1}{N}\sum_{x=1}^{N} F\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}}), \sigma^{\text{OUT}}\right) \\
&= 1 - F\left(\mathcal{M}_{N,\mathcal{U}}(\rho^{\text{IN}}), \sigma^{\text{OUT}}\right) \\
&= l_{\text{global, pure}}(\mathcal{M}_{N,\mathcal{U}}, (\rho^{\text{IN}}, \sigma^{\text{OUT}})).
\end{aligned}$$

□

Note that this inequality does not hold for the mixed loss as the Hilbert-Schmidt distance is not contractive [298]. For pure states, however, because of this lemma, we know that the local cost is automatically minimized when the global cost is minimized, but not the other way around. In that sense, the global cost is stronger than the local cost for pure states. Furthermore, using the global cost, a DQRNN can be able to learn the entanglement structure between the target outputs. Even for product targets, when employing the local cost, there is no guarantee that the global output of the DQRNN will also be a product state. If the global cost between a product state and the global output of the DQRNN is zero, however, the global output of the DQRNN will also be a product state.

On the other hand, the local cost can be easier to optimize classically and compute on NISQ devices: Numerically, before applying some tricks like scaling the learning rate, we had more problems with plateaus when classically optimizing the global cost. Although it was only

shown for shallow DQNNs (which DQRNNs are not, but they share a lot of unitaries), this falls in with sparse DQNNs (which DQRNNs are when the underlying DQNNs are not too big) not experiencing Barren plateaus when trained with a local cost [248]. On a QC, under the assumption that qudits can be reset and used again, fewer qubits are needed for the local cost than for the global cost. To compute the global cost on a quantum computer, one would have to evaluate the full DQRNN and compare the full output to the full target output using the SWAP trick. This means one would require at least $2Nm_{\text{out}}$ qudits for N iterations. To compute the local cost, we would need to repeat the experiment N times as often but require at most $2\|m\|_\infty$ qudits, which in most cases is smaller than the number of qudits needed for estimating the global cost. For each $x \in 1, ..., N$, we would ignore or reset the output in the first $x-1$ iterations and then compare the x^{th} output to the x^{th} target output. On NISQ devices, repeating the measurement is not as problematic as a high number of (logical) qubits (see Section 3.10). Hence, the local cost can be easier to optimize classically and on NISQ devices.

4.5 Classical Training Algorithm

Now, let us take a look at training algorithms on a classical computer. These can either be seen as a simulation of training DQRNNs on a QC or as using a tailored network architecture when given classical representations of quantum states. We will sketch the derivation of the algorithm here. The full derivation can be found in appendix A. We focus on a DQRNNs on qubits here, but the algorithm can easily be extended to qudits. As explained in Section 3.11.2 in more detail, for the underlying DQNN of the DQRNN, we associate a perceptron unitary U_j^l for each connection between the nodes of layer $l-1$ and node j of layer l. The unitaries acting on qudits of layer $l-1$ and l can be pooled together to a layer unitary $U^l = U_{m_l}^l \dots U_1^l$ where m_l is the number of neurons in layer l.

In the same way as for classical NNs and DQNNs, the algorithms for training DQRNNs consist of four major steps: Before training, the parameters of the network are initialized. For classical NNs, these are the weights and biases, for ff DQNNs, the perceptron unitaries, and for DQRNNs, the perceptron unitaries of the underlying DQNN. This initialization can either consist of randomly choosing parameters or loading a specific set of parameters, e.g., parameters obtained through pre-training. During training, the following three steps are repeated: First, the input is fed forward through the network. Second, some form of error, whose form depends on the cost function, is computed for the last layer and then propagated back through the network. Third, the feed-forward part and backpropagated error are used to update the parameters.

4.5.1 Local Cost with Product In- and Output

Consider the local costs

$$C_{\text{local, pure},S} = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\text{tr}(\rho_{x\,\alpha}^{\text{out}}\sigma_{x\,\alpha}^{\text{out}}) \tag{4.5.1}$$

for pure target outputs $\sigma_{x\,\alpha}^{\text{out}} = |\psi_{x\,\alpha}^{\text{out}}\rangle\langle\psi_{x\,\alpha}^{\text{out}}|$ for $\psi_{x\,\alpha}^{\text{out}} \in \mathcal{H}^{\text{out}}$ and

$$C_{\text{local, mixed},S} = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\text{tr}((\rho_{x\,\alpha}^{\text{out}} - \sigma_{x\,\alpha}^{\text{out}})^2) \tag{4.5.2}$$

for mixed target outputs. This cost makes more sense to use with product inputs and target outputs. Hence, for now, only consider product inputs and product target outputs, i.e., $\rho_\alpha^{\text{IN}} = \rho_{0\,\alpha}^{\text{mem}} \otimes \rho_{1\,\alpha}^{\text{in}} \otimes \cdots \otimes \rho_{N_\alpha\,\alpha}^{\text{in}}$, $\sigma_\alpha^{\text{OUT}} = \sigma_{1\,\alpha}^{\text{out}} \otimes \cdots \otimes \sigma_{N_\alpha\,\alpha}^{\text{out}} \otimes \sigma_{N_\alpha\,\alpha}^{\text{mem}}$, and

$$S = \{S_\alpha\}_{\alpha=1}^{M} = \left\{\left(\rho_{0\,\alpha}^{\text{mem}}, \left(\left(\rho_{x\,\alpha}^{\text{in}}, \sigma_{x\,\alpha}^{\text{out}}\right)\right)_{x=1}^{N_\alpha}, \sigma_{N_\alpha\,\alpha}^{\text{mem}}\right)\right\}_{\alpha=1}^{M}.$$

Initialization

We first have to initialize the underlying DQNN, either by loading pre-trained unitaries or by randomly choosing them. For each neuron j in layers 1 to $L+1$, we choose a unitary U_j^l on $m_{l-1}+1$ qubits at random w.r.t. the Haar measure. We then tensor it to identities on the other qubits in layer l.

Feed-forward

We know from our discussion on ff DQNNs how to feed the state on the 0^{th} layer to the last layer layer-by-layer. In each iteration of the DQRNN, the same layer-by-layer-channel is used to propagate the total input to the total output, as for the underlying DQNN (see Algorithm 9 or Equations (A.1.4) and (A.1.7)), i.e.,

$$\rho_{x\,\alpha}^l = \mathcal{E}^l\left(\rho_{x\,\alpha}^{l-1}\right) = \text{tr}^{l-1}\left(U^l\left(\rho_{x\,\alpha}^{l-1} \otimes |0\ldots0\rangle^l \langle 0\ldots0|\right)U^{l\dagger}\right) \tag{4.5.3}$$

for $l = 1, ..., l$ which leads to

$$\rho_{x\,\alpha}^{L+1} = \mathcal{N}_{\mathcal{U}}\left(\rho_{x\,\alpha}^0\right) = \text{tr}^{0:L}\left(\mathcal{U}\left(\rho_{x\,\alpha}^0 \otimes |0\ldots0\rangle^{1:L+1} \langle 0\ldots0|\right)\mathcal{U}^\dagger\right) \tag{4.5.4}$$

where we write $i : j$ for $i, ..., j$ for $i < j$. The proof for this statement can be found in appendix A.1.2 or [82, 245].

We now need to discuss how to get from one network to the next. Most generally, we tensor all inputs $\rho_{x\,\alpha}^{\text{in}}$ together and use the concatenated DQRNN on the full input ρ_α^{IN}. However, this is not very efficient. When only considering a local cost, we only compare the state $\rho_{x\,\alpha}^{\text{out}}$ to $\sigma_{x\,\alpha}^{\text{out}}$ locally instead of comparing ρ_α^{OUT} to $\sigma_{1\,\alpha}^{\text{out}} \otimes \cdots \otimes \sigma_{x\,\alpha}^{\text{out}}$. As shown in appendix A.3.2, in classical simulation, we can then write the network-to-network process more efficiently by setting

$$\rho_{x\,\alpha}^0 = \rho_{x-1\,\alpha}^{\text{mem}} \otimes \rho_{x\,\alpha}^{\text{in}} \tag{4.5.5}$$

$$\rho_{x\,\alpha}^{\text{mem}} = \text{tr}^{\text{out}}\left(\rho_{x\,\alpha}^{L+1}\right) \tag{4.5.6}$$

$$\rho_{x\,\alpha}^{\text{out}} = \text{tr}^{\text{mem}}\left(\rho_{x\,\alpha}^{L+1}\right). \tag{4.5.7}$$

for $x = 1, ..., N_\alpha$. This means we trace out the memory and output after each network to get a local output ρ_x^{out} and memory ρ_x^{mem} state, respectively. Then, the memory state is tensored to the next local input ρ_x^{in}. This is then used as input for the DQNN in the next iteration.

Backpropagation

We need to back-propagate some form of error to compute the derivative of the cost function efficiently. The backpropagation process is defined as follows:

For all $x = N_\alpha, ..., x$, we set

$$\sigma^{L+1}_{xx\,\alpha} = \sigma^{\text{out}}_{x\,\alpha} \otimes \mathbb{1}^{\text{mem}}_x \tag{4.5.8}$$

if all target outputs are pure and

$$\sigma^{L+1}_{xx\,\alpha} = 2\left(\sigma^{\text{out}}_{x\,\alpha} - \rho^{\text{out}}_{x\,\alpha}\right) \otimes \mathbb{1}^{\text{mem}}_x \tag{4.5.9}$$

if not. The difference is due to the cost being different for pure and mixed target outputs. Note that the latter is no state anymore and looks more like the error propagated through classical NNs. This is similar to our definition of σ^{L+1}_x for the DQNN (see Algorithm 9) with the addition of the identity on the memory system.

After that, we propagate $\sigma^{L+1}_{xx\,\alpha}$ not only back through the x^{th} DQNN iteration but through all iterations beforehand. For all $z = x-1, ..., 1$, we set

$$\sigma^{L+1}_{zx\,\alpha} = \sigma^{\text{mem}}_{z+1\,x\,\alpha} \otimes \mathbb{1}^{\text{out}}_z, \tag{4.5.10}$$

$$\sigma^{\text{mem}}_{zx\,\alpha} = \text{tr}^{1,\text{in}}_{z+1}\left(\left(\rho^{\text{in}}_{z+1\,\alpha} \otimes \mathbb{1}^m_z \otimes |0\dots0\rangle^1_{z+1}\langle 0\dots 0|\right) U^{1\dagger}_{z+1}\left(\mathbb{1}^0_{z+1} \otimes \sigma^1_{z+1x\,\alpha}\right) U^1_{z+1}\right). \tag{4.5.11}$$

For all $x = N_\alpha, ..., 1$, $z = x, ..., 1$, we get through each network in the same way as for the DQNN (see Algorithm 9 or Equations (A.1.9) and (A.1.5)), i.e., for $l = L, ..., 0$, we set

$$\sigma^l_{zx\,\alpha} = \mathcal{E}^{l+1^\dagger}\left(\sigma^{l+1}_{zx\,\alpha}\right) = \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right) U^{l+1^\dagger}\left(\mathbb{1}^l \otimes \sigma^{l+1}_{zx\,\alpha}\right) U^{l+1}\right). \tag{4.5.12}$$

We will need these later for updating the unitaries.

Update of Unitaries

In each training step, we update our unitaries according to

$$U^l_j \to U^{l\prime}_j = e^{i\epsilon K^l_j} U^l_j \tag{4.5.13}$$

where we want to choose the K^l_j in a way that the change in the cost function is minimized for bounded K^l_j. This way, for Hermitian K^l_j, the perceptron unitaries stay unitaries throughout the training procedure. We then find the following term for the change in the cost function in the first order.

Proposition 4.1 (Change in Local Cost). *Let $\{\mathcal{M}_{N_\alpha,\mathcal{U}}\}_\alpha$ be DQRNNs with an underlying DQNN $\mathcal{N}_\mathcal{U}$, and architecture (m^{mem}, m), where $m \in \mathbb{N}^{L+2}$, L, $m^{\text{mem}} \in \mathbb{N}$. Let the training set be of the form $S = \{S_\alpha\}^M_{\alpha=1} = \left\{\left(\rho^{\text{mem}}_{0\,\alpha}, \left(\left(\rho^{\text{in}}_{x\,\alpha}, \sigma^{\text{out}}_{x\,\alpha}\right)\right)^{N_\alpha}_{x=1}, \sigma^{\text{mem}}_{N_\alpha\,\alpha}\right)\right\}^M_{\alpha=1}$ and the $\rho^l_{x\,\alpha}, \sigma^l_{zx\,\alpha}$ be defined according to Equations (4.5.3) to (4.5.12). Performing an update on the unitaries of the form $U^l_j \to U^{l\prime}_j = e^{i\epsilon K^l_j} U^l_j$ leaves us with the updated cost*

$$C'_{\textit{local, pure/mixed},S} = 1 - \frac{1}{M}\sum_{\alpha=1}^{M} l_{\textit{local, pure/mixed}}(\mathcal{M}_{N_\alpha,\mathcal{U}'}, (\rho^{\text{IN}}_\alpha, \sigma^{\text{OUT}}_\alpha)). \tag{4.5.14}$$

We then have

$$\delta C = \lim_{\epsilon\to 0}\frac{C' - C}{\epsilon} = -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}\left(M^l_j K^l_j\right)$$

with

$$M_j^l = -\frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N}\sum_{z=1}^{x}\Big[U_j^l ... U_1^l\Big(\rho_{z\,\alpha}^{l-1}\otimes|0\dots0\rangle^l\langle 0\dots0|\Big)U_1^{l\,\dagger}\dots U_j^{l\,\dagger},$$
$$U_{j+1}^{l}{}^{\dagger} ... U_{m_l}^{l}{}^{\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_{zx\,\alpha}^l\Big)U_{m_l}^l\dots U_{j+1}^l\Big]. \tag{4.5.15}$$

Proof. See Appendices A.3.4 and A.3.6. □

The $\{M_j^l\}$ in Equation (4.5.15) are a sum over commutators between a forward and a backward part. In the forward part, we feed the inputs $\{\rho_{x\,\alpha}^{\text{in}}\}$ forward through all iterations of the underlying DQNN up to the z^{th} iteration, and forward to the $(l-1)^{\text{th}}$ layer, resulting in $\rho_{z\,\alpha}^{l-1}$. We then apply the unitaries up to neuron j, which leads to the state

$$\rho_{j\,z\,\alpha}^{l-1,l} = U_j^l ... U_1^l\Big(\rho_{z\,\alpha}^{l-1}\otimes|0\dots0\rangle^l\langle 0\dots0|\Big)U_1^{l\,\dagger}\dots U_j^{l\,\dagger}.$$

In the backward part, we start with the target output of the x^{th} iteration $\sigma_{x\,\alpha}^{\text{out}}$ (minus the x^{th} network output $\rho_{x\,\alpha}^{\text{out}}$ in the case of mixed target outputs) and go back to the z^{th} iteration (where $z \leq x$) and back to the l^{th} layer, leading to $\sigma_{zx\,\alpha}^l$. We then apply the adjoint unitaries back to neuron $j+1$, which results in

$$\sigma_{j+1\,zx\,\alpha}^{l-1,l} = U_{j+1}^{l}{}^{\dagger} ... U_{m_l}^{l}{}^{\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_{zx\,\alpha}^l\Big)U_{m_l}^l\dots U_{j+1}^l.$$

The $\{M_j^l\}$ are then calculated as a sum over the commutator between the feed-forward and the backpropagated part via

$$M_j^l = -\frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N}\sum_{z=1}^{x}\Big[\rho_{j\,z\,\alpha}^{l-1,l}, \sigma_{j+1\,zx\,\alpha}^{l-1,l}\Big].$$

Note that the matrix M_j^l is defined in the same way for mixed and pure target outputs although the cost is different, because we also chose the starting state in the backward process slightly differently for mixed and pure target outputs.

As δC is linear in K_j^l, it would be minimized by an infinite matrix [82]. However, we want the K_j^l to be bounded, so we have to introduce a constraint, which results in the following proposition.

Proposition 4.2 ([82]). *Let* $L \in \mathbb{N}$, $m \in \mathbb{N}^{L+2}$, $K_j^l \in \mathbb{C}^{(m_{l-1}+1)\times(m_{l-1}+1)}$, $M_j^l \in \mathbb{C}^{(m_{l-1}+m_l)\times(m_{l-1}+m_l)}$ *for* $j = 1, ..., m_l$, $l = 1, ..., L+1$. *The minimization problem*

$$\min -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}(M_j^l(\mathbb{1}_{1:j-1}^l\otimes K_j^l\otimes\mathbb{1}_{j+1:m_l}^l))$$
$$K_j^l = \sum_{\vec{\gamma}} K_{j\,\vec{\gamma}}^l\sigma^{\gamma_1}\otimes\dots\sigma^{\gamma_{m_{l-1}+1}}$$
$$\sum_{\vec{\gamma}} {K_{j\,\vec{\gamma}}^l}^2 = const.$$

has the solution

$$K_j^l = i\frac{2^{m_{l-1}}i}{\lambda}\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big)$$

where λ is a Lagrange multiplier introduced to solve the constrained minimization process and the σ^{γ_i} are the Pauli matrices.

Proof. The proof can be found in [82] or Appendix A.1.4. □

Hence, in each training step, we perform the update

$$U_j^l \mapsto \exp\left(P_j^l\right) U_j^l$$

with

$$P_j^l = i\epsilon K_j^l = -2^{m_{l-1}}\eta\,\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big),$$

where $\eta = \frac{\epsilon}{\lambda}$ is the learning rate.

The algorithm

Together, this leaves us with Algorithm 10 where we write $U_{j_1:j_2}^l = U_{j_2}^l \dots U_{j_1}^l$ for $j_1 \leq j_2$, and $U_{j_1:j_2}^l = \mathbb{1}$ for $j_1 > j_2$. The algorithm uses the training set $S = ((\rho_{0\,\alpha}^{\mathrm{mem}}, (\rho_{x\,\alpha}^{\mathrm{in}}, \sigma_{x\,\alpha}^{\mathrm{out}}))_x)_\alpha$, the learning rate η, the number of training steps T, and either a DQRNN architecture (m, m^{mem}) or a list of perceptron unitaries $((U_j^l)_j)_l$ and the number of memory qubits (m^{mem}) as input. In the first step, the unitaries are randomly initialized, or the architecture calculated, if not already given. Then, in each training step, the inputs are fed forward through the DQRNN. Afterward, the error, depending on the cost, is calculated in the last layer of each DQNN iteration and propagated back through the full DQRNN. For each $l = 1, ..., L+1,\ j = 1, ..., m_l$, we then compute an update-matrix P_j^l. It is essentially a sum over training pairs $\alpha = 1, ..., M$, and DQNN iterations $x = 1, ..., N,\ z = 1, ..., x$ of the partial trace of a commutator of feeding the input forward to neuron j in layer l of DQNN iteration z and propagating the error back from the output of DQNN iteration x to neuron $j+1$ in layer l of DQNN iteration z. The unitaries are then updated as $U_j^l \to e^{\eta P_j^l} U_j^l$. To keep track of the progress, the cost is calculated in each training step.

Note that the size of the used matrices scales only with the width, not the depth or number of iterations of the underlying DQNN.

The here presented algorithm corresponds to the vanilla gradient descent algorithm for classical NNs. It can easily be generalized to include adjustments like mini-batches, validation data, or early-stopping conditions.

4.5.2 Global Cost and Pure Target Outputs

Now, consider the global cost

$$C_{\text{global, pure},S} = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\mathrm{tr}(\rho_\alpha^{\mathrm{OUT}}\sigma_\alpha^{\mathrm{OUT}})$$

Algorithm 10 Classical Training Algorithm of DQRNNs with and product in- and output local cost

Input: $((\rho^{\text{mem}}_{0\,\alpha}, (\rho^{\text{in}}_{x\,\alpha}, \sigma^{\text{out}}_{x\,\alpha}))_{x=1,\dots,N})_{\alpha=1,\dots,M}$, m_{mem}, $(m_l)_{l=0,\dots,L+1}$ or $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$LEN$(((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$LEN$((U^l_j)_{j=1,\dots,m_l})$

  $L \leftarrow$LEN$((m_l)_{l=1,\dots,L+1})-1$

else if $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      $U^l_j \leftarrow$RANDOMUNITARYHAAR$(m_{l-1}+1$ qudits$)$ ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits*

      $U^l_j \leftarrow \mathbb{1}^l_{1:j-1} \otimes U^l_j \otimes \mathbb{1}^l_{j+1:m_l}$

for t=1,...,T **do** ▷ *In each training step*

  for $\alpha = 1, \dots, M$ **do** ▷ *Feed-forward*

    for x=1,...,N **do**

      $\rho^0_{x\,\alpha} \leftarrow \rho^{\text{mem}}_{x-1\,\alpha} \otimes \rho^{\text{in}}_{x\,\alpha}$

      for l=1,...,L+1 **do**

        $\rho^l_{x\,\alpha} \leftarrow \text{tr}^{l-1}\left(U^l \left(\rho^{l-1}_{x\,\alpha} \otimes |0\dots0\rangle^l \langle 0\dots0|\right) U^{l\dagger}\right)$

      $\rho^{\text{mem}}_{x\,\alpha} \leftarrow \text{tr}^{\text{out}}\left(\rho^{L+1}_{x\,\alpha}\right)$

      $\rho^{\text{out}}_{x\,\alpha} \leftarrow \text{tr}^{\text{mem}}\left(\rho^{L+1}_{x\,\alpha}\right)$

  for $\alpha = 1, \dots, M$ **do** ▷ *Backpropagation*

    for $x = 1, \dots, N_\alpha$ **do**

      if $\forall\, \alpha \in \{1:M\}, x \in \{1:N_\alpha\}\ \exists\, \psi^{\text{out}}_{x\,\alpha} \in \mathcal{H}^{\text{in}}$ such that $\sigma^{\text{out}}_{x\,\alpha} = |\psi^{\text{out}}_{x\,\alpha}\rangle \langle\psi^{\text{out}}_{x\,\alpha}|$ **then**

        $\sigma^{L+1}_{xx\,\alpha} \leftarrow \sigma^{\text{out}}_{x\,\alpha} \otimes \mathbb{1}^{\text{mem}}$

      else

        $\sigma^{L+1}_{xx\,\alpha} \leftarrow 2\left(\sigma^{\text{out}}_{x\,\alpha} - \rho^{\text{out}}_{x\,\alpha}\right) \otimes \mathbb{1}^{\text{mem}}_x$

      for $z = x, \dots, 1$ **do**

        for $l = L, \dots, 0$ **do**

          $\sigma^l_{zx\,\alpha} \leftarrow \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right) U^{l+1\dagger} \left(\mathbb{1}^l \otimes \sigma^{l+1}_{zx\,\alpha}\right) U^{l+1}\right)$

        $\sigma^{\text{mem}}_{z-1\,x\,\alpha} \leftarrow \text{tr}^{1,\text{in}}_z\left(\left(\rho^{\text{in}}_{z\,\alpha} \otimes \mathbb{1}^m_{z-1} \otimes |0\dots0\rangle^1_z \langle 0\dots0|\right) U^{1\dagger}_z \left(\mathbb{1}^0_z \otimes \sigma^1_{zx\,\alpha}\right) U^1_z\right)$

        $\sigma^{L+1}_{z-1\,x\,\alpha} \leftarrow \mathbb{1}^{\text{out}} \otimes \sigma^{\text{mem}}_{z-1\,x\,\alpha}$

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      $$M^l_j \leftarrow \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\sum_{z=1}^{x}\Big[U^l_{1:j}\left(\rho^{l-1}_{x\,\alpha} \otimes |0\dots0\rangle^l \langle 0\dots0|\right)U^{l\,\dagger}_{1:j}, U^l_{j+1:m_l}\left(\mathbb{1}^{l-1} \otimes \sigma^l_{zx\,\alpha}\right)U^{l\,\dagger}_{j+1:m_l}\Big]$$

      $P^l_j \leftarrow -2^{m_{l-1}}\eta\,\text{tr}^l_{1:j-1}\left(\text{tr}^l_{j+1:m_l}\left(M^l_j\right)\right)$

      $U^l_j \leftarrow \exp\left(P^l_j\right) U^l_j$

  ▷ *Calculate cost* ◁

  if $\forall\, \alpha \in \{1:M\}, x \in \{1:N_\alpha\}\ \exists\, \psi^{\text{out}}_{x\,\alpha} \in \mathcal{H}^{\text{in}}$ such that $\sigma^{\text{out}}_{x\,\alpha} = |\psi^{\text{out}}_{x\,\alpha}\rangle \langle\psi^{\text{out}}_{x\,\alpha}|$ **then**

    $C_t \leftarrow 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\text{tr}\left(\rho^{\text{out}}_{x\,\alpha}\sigma^{\text{out}}_{xx\,\alpha}\right)$

  else

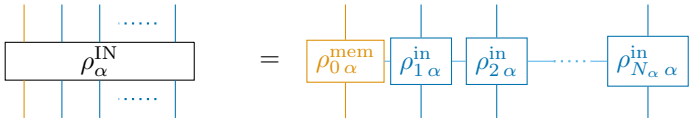
    $C_t \leftarrow \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\text{tr}\left(\left(\rho^{\text{out}}_{x\,\alpha} - \sigma^{\text{out}}_{xx\,\alpha}\right)^2\right)$

for pure target outputs $\sigma_\alpha^{\text{OUT}} = |\psi_\alpha^{\text{OUT}}\rangle\langle\psi_\alpha^{\text{OUT}}|$ for $\psi_\alpha^{\text{OUT}} \in \mathcal{H}_{N_\alpha}^{\text{OUT}}$. The global cost is not only useful for analyzing product data but also for data that is entangled between different iterations of the underlying DQNN. Hence, we drop the assumption that the in- and outputs of the DQRNN are product states. Then, as discussed before, the training set is, in general, of the form

$$S = \left\{\left(\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}}\right)\right\}_{\alpha=1}^{M} \in \bigtimes_{\alpha=1}^{M} \left(\mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{IN}}) \times \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{OUT}})\right)$$

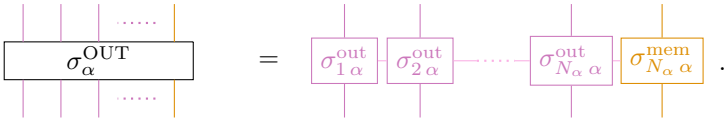
if we do M runs of the DQRNN and iterate over the underlying DQNN N_α times.

In this case, we have to handle – on potentially many sites – entangled states, which can result in high dimensional matrices. Hence, it makes sense to use TNN for computations and the derivation of the training algorithm. Every entangled state can be written as a matrix product operator (MPO), so we can write



(4.5.16)

and



.

(4.5.17)

Here, and in the following, we use different colors for the input (blue), memory (orange), output (purple), and hidden (green) layers. For now, consider only pure target output states $\sigma_\alpha^{\text{OUT}} = |\psi_\alpha^{\text{OUT}}\rangle\langle\psi_\alpha^{\text{OUT}}|$ for $\psi_\alpha^{\text{OUT}} \in \mathcal{H}_{N_\alpha}^{\text{OUT}}$.

The output of the network is then given by

$$\begin{aligned}\rho_\alpha^{\text{OUT}} &= \left(\mathcal{N}_{\mathcal{U}_{N_\alpha}} \circ \cdots \circ \mathcal{N}_{\mathcal{U}_1}\right)\left(\rho^{\text{IN}}\right) = \left(\mathcal{N}_{\mathcal{U}_N} \circ \cdots \circ \mathcal{N}_{\mathcal{U}_1}\right)\left(\rho^{\text{IN}}\right) \\ &= \text{tr}_{1:N_\alpha}^{0:L}\left(\mathcal{U}_{N_\alpha} \ldots \mathcal{U}_1 \left(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N_\alpha}^{1:L+1}\langle 0\ldots0|\right)\mathcal{U}_1^\dagger \ldots \mathcal{U}_{N_\alpha}^\dagger\right).\end{aligned}$$

For the output in TNN, see Figure 4.2.

We can then write the cost function

$$\begin{aligned}C_{\text{global, pure},S} &= 1 - \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}(\rho_\alpha^{\text{OUT}}\sigma_\alpha^{\text{OUT}}) \\ &= 1 - \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}\Big(\left(\sigma_\alpha^{\text{OUT}} \otimes \mathbb{1}_{1:N_\alpha}^{0:L}\right)\mathcal{U}_{N_\alpha} \ldots \mathcal{U}_1 \left(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N_\alpha}^{1:L+1}\langle 0\ldots0|\right) \\ &\quad \mathcal{U}_1^\dagger \ldots \mathcal{U}_{N_\alpha}^\dagger\Big)\end{aligned}$$

in TNN, as depicted in Figure 4.3 which is the sum of the trace of the product of the output with the target output. Hence, in TNN, we only connect the red lines in Figure 4.2 with the ones in Equation (4.5.17) and sum over the result.

The training algorithm is then derived using the same update as before, and careful bubbling. The perceptron unitaries are again initialized as before, and the training procedure again

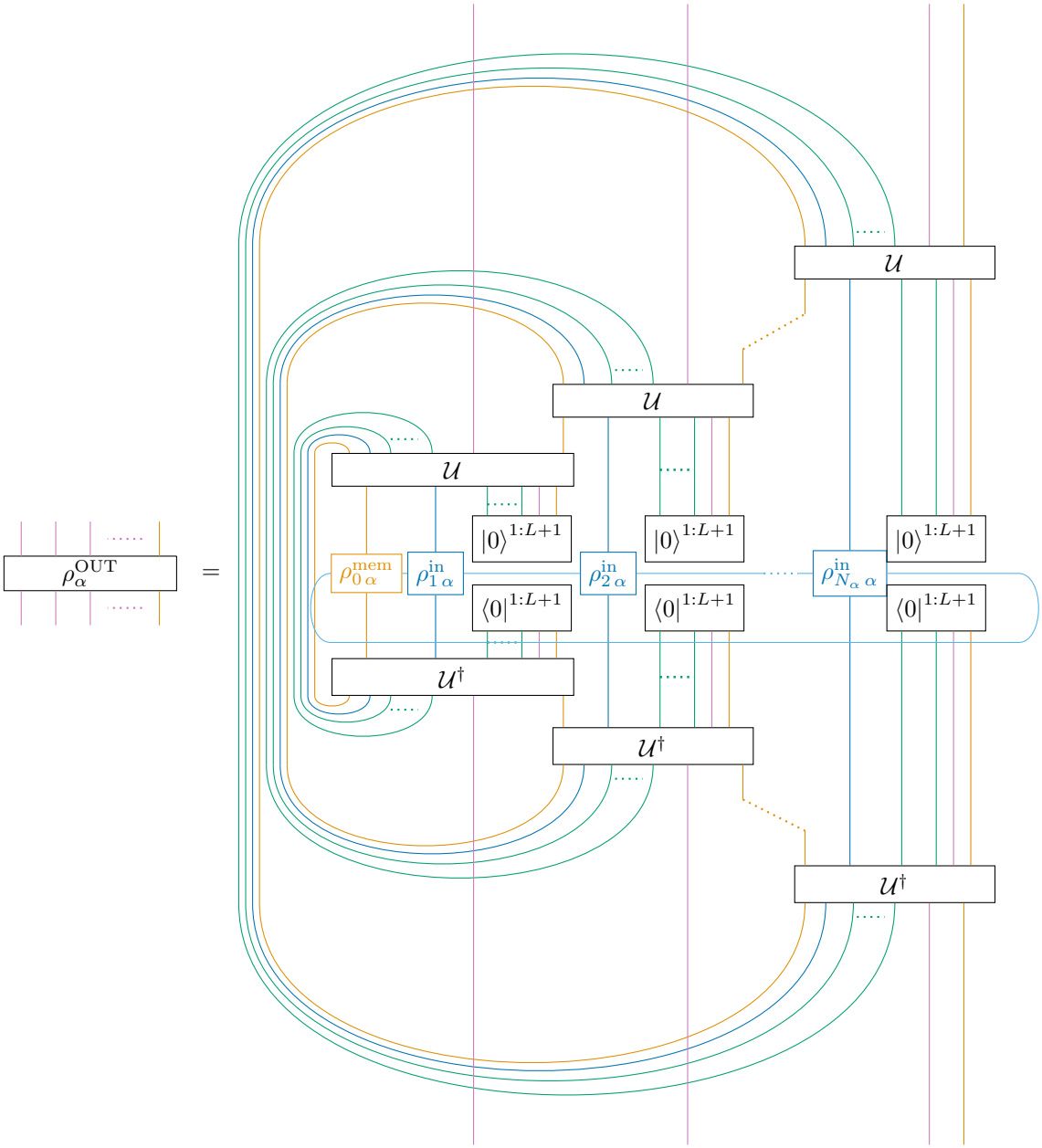


Figure 4.2: **Network output of a DQRNN** with N_α iterations over the underlying DQNN with input ρ^{IN} as given in Equation (4.5.16).

$C = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}$

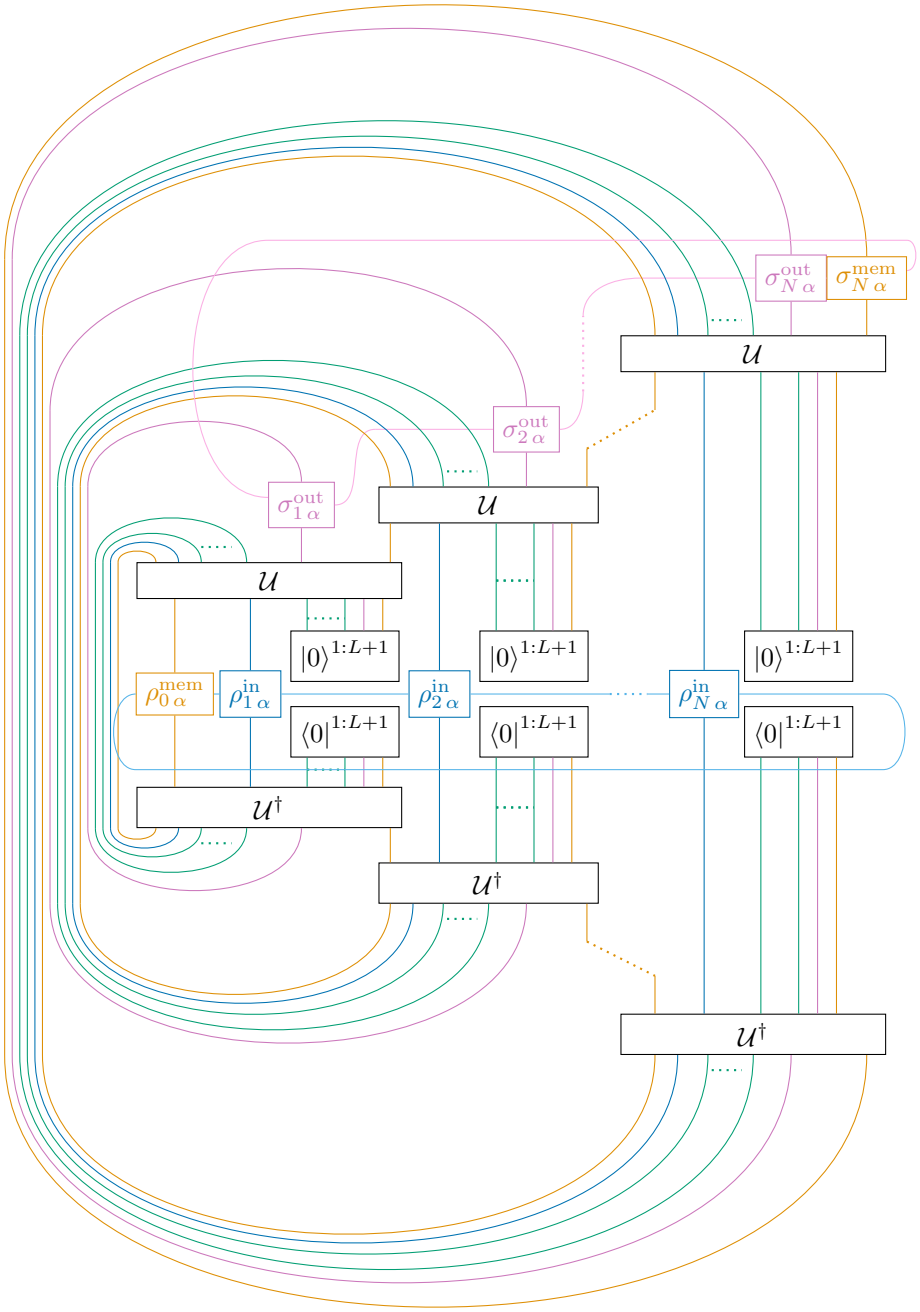


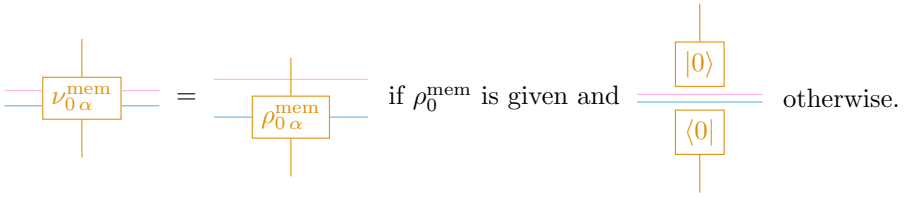
Figure 4.3: **Global Cost of a DQRNN** for global inputs and target outputs given as detailed in Equations (4.5.16) and (4.5.17).

consists of a feed-forward, a backpropagation, and an update part. The derivation of the training algorithm can be found in Appendix A.4.

Feed-forward

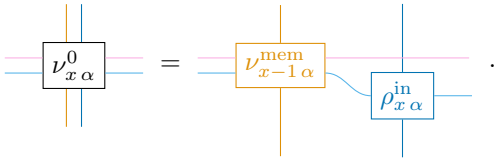
We define our way forward similarly to before with two differences: First, we have to keep track of the virtual legs. Second, we modify the way from one iteration of the underlying DQNN to the next, as we do not want to throw away the outputs.

To simplify the calculation of the cost, for all $\alpha = 1, ..., M$, set



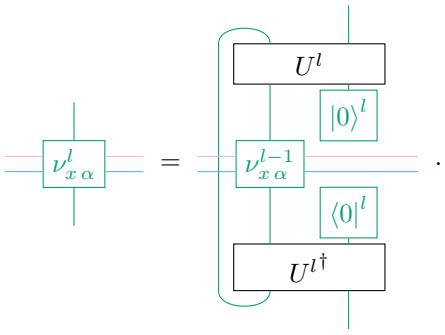
(4.5.18)

For $x = 1, ..., N_\alpha$, set



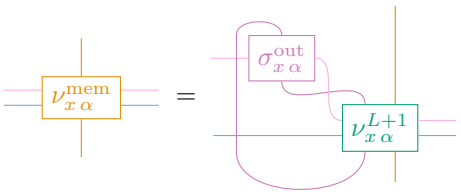
(4.5.19)

For $l = 1, ..., L+1$, set



(4.5.20)

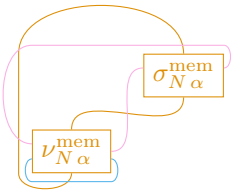
Until now, this is the same as for the local cost if we had kept track of the virtual legs. However, in order to not have to keep every output leg open until the last iteration, which would result in a matrix of size Nm^{out}, we already group the x^{th} target output and the last layer output of the x^{th} iteration of the underlying DQNN together, i.e.,



(4.5.21)

This means that this is not the forward propagated state but some kind of error depending on the cost, as in the backpropagation part of the local cost. The cost is then given by

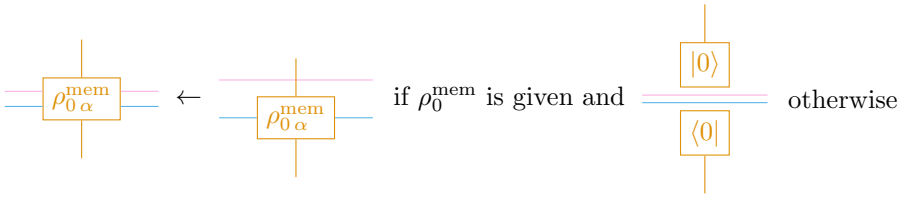
$$C = 1 - \tfrac{1}{M} \sum_{\alpha=1}^{M}$$



(4.5.22)

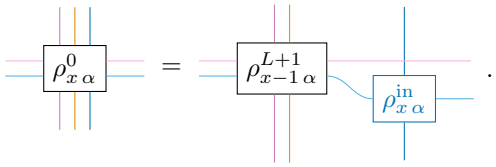
as further explained in appendix A.4.1.

Note that Equations (4.5.19) and (4.5.21), especially, are only useful if we want to calculate the cost. The full output state will not arise from this feed-forward procedure. To obtain the full output, we instead have to skip Equation (4.5.21) and use $\nu^{L+1}_{x-1\,\alpha}$ instead of $\nu^{\text{mem}}_{x-1\,\alpha}$ in Equation (4.5.19). We refer to the corresponding states as $\rho^{l}_{x\,\alpha}$ and set



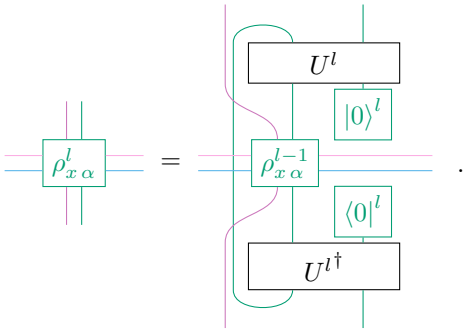
(4.5.23)

for $\alpha = 1, ..., M$. For notational clearance, we abuse notation and set $\rho_0^{L+1} = \rho_0^{\text{mem}}$ although there is no 0^{th} iteration and ρ_0^{L+1} is not acting on $\mathcal{H}^{\text{out}}$. In the next two equations, the output legs also only appear for $x > 1$. They are nevertheless depicted to keep things shorter. For $x = 1, ..., N_\alpha$, set



(4.5.24)

Furthermore, this and the following states are not only acting on one layer of the DQNN iteration x, but also on the $(0 : x-1)^{\text{th}}$ output layer. For $l = 1, ..., L+1$, set

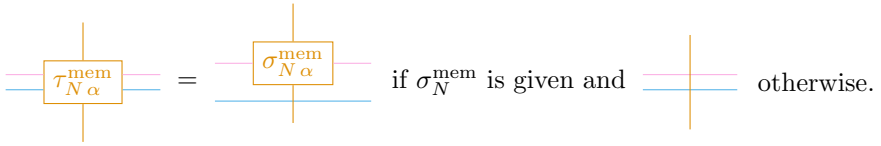


(4.5.25)

Following these equations, we would get the output state, but we would have to use states of up to $Nm^{\text{out}} + \|m\|_\infty$ qubits. So, the computation of these states and computations with them would not be efficient anymore. In particular, the size of the matrices would scale exponentially with the number of iterations N over the underlying DQNN as $2^{m^{\text{out}}N}$.

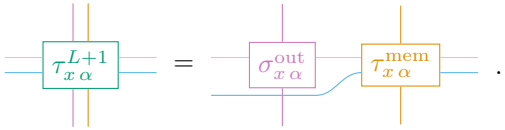
Backpropagation

The backpropagation again mimics the one of the local case. For later calculating the update matrices, it is useful to define define, for $\alpha = 1, ..., M$,



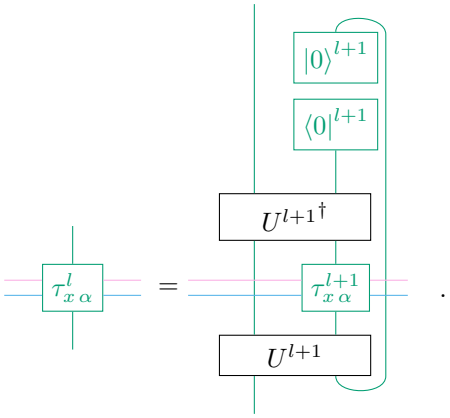
(4.5.26)

For $x = N_\alpha, ..., 1$, set



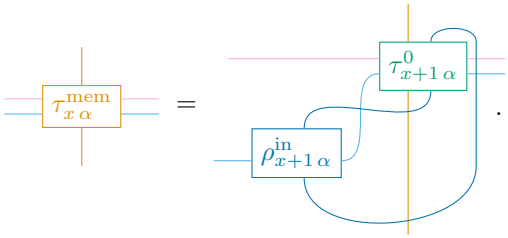
(4.5.27)

For $l = L, ..., 0$, set



(4.5.28)

Set



(4.5.29)

Update

In each training step, our unitaries again get updated according to

$$U^l_j \to U^{l\prime}_j = e^{i\epsilon K^l_j} U^l_j \tag{4.5.30}$$

where we choose the K^l_j in a way that the change in cost function is minimized for bounded K^l_j.

Proposition 4.3 (Change in Global Cost). *Let $\mathcal{M}_{N_\alpha,\mathcal{U}}$ be DQRNNs with $N \in \mathbb{N}$ iterations over an underlying DQNN $\mathcal{N}_\mathcal{U}$, and architecture (m^{mem}, m), where $m \in \mathbb{N}^{L+2}$, L, $m^{\text{mem}} \in \mathbb{N}$. Let the training set be of the form $S = \left\{\left(\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}}\right)\right\}_{\alpha=1}^{M} \in \times_{\alpha=1}^{M}\left(\mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{IN}}) \times \mathcal{D}(\mathcal{H}_{N_\alpha}^{\text{OUT}})\right)$ with pure $\sigma_\alpha^{\text{OUT}}$, where the ρ_α^{IN}, $\sigma_\alpha^{\text{OUT}}$ are written as in Equations (4.5.16) and (4.5.17). We define $\nu_{x\,\alpha}^l, \tau_{x\,\alpha}^l$ according to Equations (4.5.18) to (4.5.21), and (4.5.26) to (4.5.29). Performing an update on the unitaries of the form $U_j^l \to U_j^{l\prime} = e^{i\epsilon K_j^l} U_j^l$ leaves us with the updated cost*

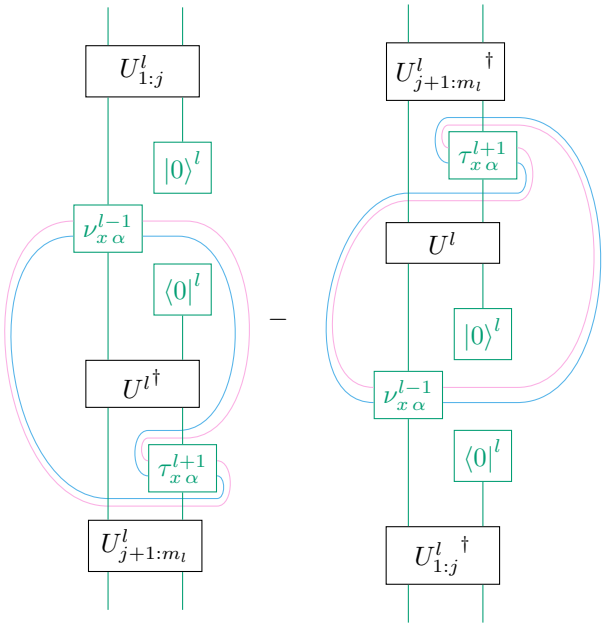
$$C'_{\textit{global, pure},S} = 1 - \frac{1}{M}\sum_{\alpha=1}^{M} l_{\textit{global, pure}}(\mathcal{M}_{N_\alpha,\mathcal{U}'}, (\rho_\alpha^{\text{IN}}, \sigma_\alpha^{\text{OUT}})). \tag{4.5.31}$$

We then have

$$\delta C = \lim_{\epsilon\to 0}\frac{C' - C}{\epsilon} = -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}\left(M_j^l K_j^l\right)$$

with

$$M_j^l = -\tfrac{1}{M}\textstyle\sum_{\alpha=1}^{M}\sum_{x=1}^{N_\alpha}$$



(4.5.32)

Proof. See Appendix A.4.3. □

The M_j^l in Equation (4.5.32) are again commutators between a forward and a backward part and derived by bubbling TNs. In the forward part, we feed the input error forward through all iterations of the underlying DQNN up to the x^{th} iteration and forward to the $(l-1)^{\text{th}}$ layer. We then apply the unitaries in layer l up to neuron j. In the backward part, we start with the output of the N^{th} iteration and go back to the x^{th} iteration and back to the l^{th} layer. We then apply the adjoint unitaries back to neuron $j+1$.

Optimizing δC w.r.t. K_j^l under the constraint that K_j^l is bounded, using Proposition 4.2, again yields

$$K_j^l = i\frac{2^{m_{l-1}}i}{\lambda}\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big)$$

where λ is again the Lagrange multiplier introduced to solve the constrained minimization process. Hence, in each training step, we perform the update

$$U_j^l \mapsto \exp\left(P_j^l\right) U_j^l$$

with

$$P_j^l = i\epsilon K_j^l = -2^{m_{l-1}}\eta\,\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big),$$

where $\eta = \frac{\epsilon}{\lambda}$ is the learning rate.

The Algorithm

The initialization is performed in the same way as discussed for the local cost. This yields Algorithm 11. Again, the algorithm uses the training set $S = ((\rho_{0\,\alpha}^{\mathrm{mem}}, (\rho_{x\,\alpha}^{\mathrm{in}}, \sigma_{x\,\alpha}^{\mathrm{out}}))_x)_\alpha$ (now consisting of MPS representations), the learning rate η, the number of training steps T, and either a DQRNN architecture (m, m^{mem}) or a list of perceptron unitaries $((U_j^l)_j)_l$ and the number of memory qubits (m^{mem}) as input. In the first step, the unitaries are randomly initialized, or the architecture calculated, if not already given. Then, in each training step, one kind of error is computed in the input layer of the first DQNN iteration and fed forward through the DQRNN. Afterward, another kind of error is calculated in the last layer of the last DQNN iteration and propagated back through the DQRNN. For each j, l, we then compute an update-matrix P_j^l, which is essentially the trace of a commutator of feeding one error forward to neuron j in layer l and propagating the other error back to neuron $j+1$ in layer l. The unitaries are then updated as $U_j^l \to e^{\eta P_j^l} U_j^l$. To keep track of the progress, the cost is calculated in each training step.

Note that the size of the matrices used in this algorithm scales with the bond dimension of the global input and output MPS and the width of the underlying DQNN but not its depth or the number of iterations.

Product data

If the input and target output of our training set are product states, we can simply ignore the virtual legs in Algorithm 11. Then, the size of the matrices used in this algorithm scales with the width of the underlying DQNN but not its depth or the number of iterations.

During our initial numerical experiments with $N = 20$, with a constant learning rate, we struggled with a plateauing cost at the start of training or oscillations in the cost at the end of training. This is because the M_j^l can be very small if $\rho_\alpha^{\mathrm{OUT}}$ and $\sigma_{1\,\alpha}^{\mathrm{out}} \otimes \cdots \otimes \sigma_{N\,\alpha}^{\mathrm{out}} \otimes \sigma_{N\,\alpha}^{\mathrm{mem}}$ are far away from each other, i.e., the cost is close to one. However, during training, when the $\rho_\alpha^{\mathrm{OUT}}$ and $\sigma_{1\,\alpha}^{\mathrm{out}} \otimes \cdots \otimes \sigma_{N\,\alpha}^{\mathrm{out}} \otimes \sigma_{N\,\alpha}^{\mathrm{mem}}$ are closer together, i.e., when the cost is closer to zero, the size of the M_j^l increases. This falls in line with cost function landscapes getting exponentially flatter and minima getting narrower for a global cost [248, 254]. Hence, with a constant learning rate, updates at the start of training would be very small and become larger during training. However, in view of our discussion of the learning rate in Section 2.1.5,

Algorithm 11 Classical Training Algorithm of DQRNNs with global cost, MPS data and pure target output

Input: $((\rho^{\text{mem}}_{0\,\alpha}, ((\rho^{\text{in}}_{x\,\alpha}, \sigma^{\text{out}}_{x\,\alpha}))_{x=1,\dots,N}, \sigma^{\text{mem}}_{N\,\alpha}))_{\alpha=1,\dots,M}$, where $\sigma^{\text{out}}_{x\,\alpha}$ pure, $(m_l)_{l=0,\dots,L+1}, m_{\text{mem}}$ or $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U^l_j)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits* ◁

      $U^l_j \leftarrow$RandomUnitaryHaar($m_{l-1}+1$ qudits)

      $U^l_j \leftarrow \mathbb{1}^l_{1:j-1} \otimes U^l_j \otimes \mathbb{1}^l_{j+1:m_l}$

for t=1,...,T **do** ▷ *In each training step*

  for $\alpha = 1, \dots, M$ **do** ▷ *Feed-forward*

    Calculate $\nu^{\text{mem}}_{0\,\alpha}$ according to Equation (4.5.18)

    for x=1,...,N **do**

      Calculate $\nu^0_{x\,\alpha}$ according to Equation (4.5.19)

      for l=1,...,L+1 **do**

        Calculate $\nu^l_{x\,\alpha}$ according to Equation (4.5.20)

      Calculate $\nu^{\text{mem}}_{x\,\alpha}$ according to Equation (4.5.21)

  for $\alpha = 1, \dots, M$ **do** ▷ *Backpropagation*

    Calculate $\tau^{\text{mem}}_{N_\alpha\,\alpha}$ according to Equation (4.5.26)

    for $x = 1, \dots, N_\alpha$ **do**

      Calculate $\tau^{L+1}_{x_\alpha\,\alpha}$ according to Equation (4.5.27)

      for $l = 0, \dots, L$ **do**

        Calculate $\tau^l_{x_\alpha\,\alpha}$ according to Equation (4.5.28)

      Calculate $\tau^{\text{mem}}_{x_\alpha\,\alpha}$ according to Equation (4.5.29)

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      Calculate M^l_j according to Equation (4.5.32)

      $P^l_j \leftarrow -2^{m_{l-1}} \eta \, \text{tr}^l_{1:j-1}\left(\text{tr}^l_{j+1:m_l}\left(M^l_j\right)\right)$

      $U^l_j \leftarrow \exp\left(P^l_j\right) U^l_j$

  Calculate C_t according to Equation (4.5.22) ▷ *Calculate cost*

this is the opposite of what we usually want to do. The updates should get smaller during training. In our numerical results, it helped to rescale the learning rate to $\tilde{\eta} = \frac{\eta}{1-C}$, where η is constant. This makes the learning rate larger for big costs and pulls the size of the updates closer together. Maximizing $\ln(1-C)$, which is equivalent to minimizing C, instead of minimizing C results directly in the modified update rule. The update rule then changes to

$$P_j^l = -2^{m_{l-1}} \frac{\eta}{1-C_t} \operatorname{tr}_{1:j-1}^l \left(\operatorname{tr}_{j+1:m_l}^l \left(M_j^l \right) \right) \tag{4.5.33}$$

which leads to Algorithm 12.

4.6 Numerical Results

All numerics were performed on up to 3 qubits per layer, product data, up to $N = 100$ iterations over the underlying DQNN, and $M = 1$.

4.6.1 Local Cost and Product Data with Pure Target Output

Most numerical experiments were performed using the local cost, product training data, and pure target outputs. The number of channels that do not entangle the different outputs are very limited, which is why we focus on only two examples here.

Delay Channel

The first example is a standard example used when talking about quantum channels with memory: the delay channel we already discussed in Example 3.4. Before discussing training a DQRNN on the delay channel, let us review it. We obtain the delay channel by using the SWAP gate as the underlying channel of a quantum channel with memory, where we, in each step, swap the input/output with part of the memory system. This results in the first outputs being the initial memory qubits and the later ones being the earlier inputs. We use $\mathcal{H}^{\text{in}} = \mathbb{C}^2, \mathcal{H}^{\text{out}} = \mathbb{C}^2, \mathcal{H}^{\text{mem}} = \left(\mathbb{C}^2\right)^k$. Given an input state

$$\rho^{\text{IN}} = (\rho_0^{\text{mem}})_1 \otimes \cdots \otimes (\rho_0^{\text{mem}})_k \otimes \rho_1^{\text{in}} \otimes \cdots \otimes \rho_N^{\text{in}},$$

the output and memory of the delay-by-k channel after N iterations will be

$$\sigma^{\text{OUT}} = \sigma_1^{\text{out}} \otimes \cdots \otimes \sigma_N^{\text{out}} \otimes (\sigma_N^{\text{mem}})_1 \otimes \cdots \otimes (\sigma_N^{\text{mem}})_k,$$

where

$$\sigma_x^{\text{out}} = \begin{cases} (\rho_0^{\text{mem}})_x, & \text{if } x \leq k, \\ \rho_{x-k}^{\text{in}}, & \text{if } k < x \leq N, \end{cases}$$
$$(\sigma_N^{\text{mem}})_x = \rho_{N-k+x}^{\text{in}} \quad \text{for } x = 1, ..., k,$$

and we assumed the different inputs and also the initial memory to be in a product state. We can use this to generate training and testing data sets. First, the input states are chosen at random w.r.t. the Haar measure discussed in Section 3.8 as pure states. Then, we apply the delay-by-k channel to generate outputs. This results in sets of the form

$$S = \left(\left|\phi_0^{\text{mem}}\right\rangle, \left(\left|\phi_1^{\text{in}}\right\rangle, \left|(\phi_0^{\text{mem}})_1\right\rangle \right), \ldots, \left(\left|\phi_x^{\text{in}}\right\rangle, \left|\phi_{x-k}^{\text{in}}\right\rangle \right), \ldots, \left(\left|\phi_N^{\text{in}}\right\rangle, \left|\phi_{N-k}^{\text{in}}\right\rangle \right) \right)$$

which are then used to train DQRNNs and ff DQNNs. For this example, it is clear that a

Algorithm 12 Classical Training Algorithm of DQRNNs with global cost, product data, and pure target outputs

Input: $((\rho^{\text{mem}}_{0\,\alpha}, ((\rho^{\text{in}}_{x\,\alpha}, \sigma^{\text{out}}_{x\,\alpha}))_{x=1,\dots,N}, \sigma^{\text{mem}}_{N\,\alpha}))_{\alpha=1,\dots,M}$, $(m_l)_{l=0,\dots,L+1}, m_{\text{mem}}$ or $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U^l_j)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U^l_j)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits* ◁

      $U^l_j \leftarrow$RandomUnitaryHaar$(m_{l-1}+1$ qudits$)$

      $U^l_j \leftarrow \mathbb{1}^l_{1:j-1} \otimes U^l_j \otimes \mathbb{1}^l_{j+1:m_l}$

for t=1,...,T **do** ▷ *In each training step*

  for $\alpha = 1, \dots, M$ **do** ▷ *Feed-forward*

    $\nu^{\text{mem}}_{0\,\alpha} \leftarrow \rho^{\text{mem}}_{0\,\alpha}$

    for x=1,...,N **do**

      $\nu^0_{x\,\alpha} \leftarrow \rho^{\text{in}}_{x\,\alpha} \otimes \nu^{\text{mem}}_{x-1\,\alpha}$

      for l=1,...,L+1 **do**

        $\nu^l_{x\,\alpha} \leftarrow \text{tr}_{l-1}\left(U^l(\nu^{l-1}_{x\,\alpha} \otimes |0\dots0\rangle_l \langle 0\dots0|)U^{l\dagger}\right).$

      $\nu^{\text{mem}}_{x\,\alpha} \leftarrow \text{tr}^{\text{out}}\left((\sigma^{\text{out}}_{x\,\alpha} \otimes \mathbb{1}^{\text{mem}}_x)\nu^{L+1}_{x\,\alpha}\right)$

  for $\alpha = 1, \dots, M$ **do** ▷ *Backpropagation*

    Calculate $\tau^{\text{mem}}_{N_\alpha\,\alpha} \leftarrow \sigma^{\text{mem}}_N$

    for $x = 1, \dots, N_\alpha$ **do**

      $\tau^{L+1}_{x\,\alpha} \leftarrow \tau^{\text{mem}}_{x\,\alpha} \otimes \sigma^{\text{out}}_{x\,\alpha}$

      for $l = L, \dots, 0$ **do**

        $\tau^l_{x\,\alpha} \leftarrow \text{tr}_{l+1}\left((\mathbb{1}^l \otimes |0\dots0\rangle_{l+1}\langle 0\dots0|)U^{l+1\dagger}(\mathbb{1}^l \otimes \tau^{l+1}_{x\,\alpha})U^{l+1}\right)$

      Calculate $\tau^{\text{mem}}_{x-1\,\alpha} \leftarrow \text{tr}^{\text{in}}\left((\mathbb{1}^{\text{mem}}_{x-1} \otimes \rho^{\text{in}}_{x\,\alpha})\tau^0_{x\,\alpha}\right)$

  $C_t \leftarrow 1 - \frac{1}{M}\sum_{\alpha=1}^{M} \text{tr}\left(\rho^{\text{mem}}_{N_\alpha}\right)$ ▷ *Calculate cost*

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

$$M^l_j = \frac{1}{M}\sum_{\alpha=1}^{M}\sum_{x=1}^{N}\Big[U^l_{1:j}\left(\nu^{l-1}_{x\,\alpha} \otimes |0\dots0\rangle_l\langle 0\dots0|\right)U^{l\dagger}_{1:j},\; U^l_{j+1:m_l}{}^{\dagger}\left(\mathbb{1}^{l-1} \otimes \tau^l_{x\,\alpha}\right)U^l_{j+1:m_l}\Big]$$

      $P^l_j \leftarrow -2^{m_{l-1}}\frac{\eta}{1-C_t}\,\text{tr}^l_{1:j-1}\left(\text{tr}^l_{j+1:m_l}\left(M^l_j\right)\right)$

      $U^l_j \leftarrow \exp\left(P^l_j\right)U^l_j$

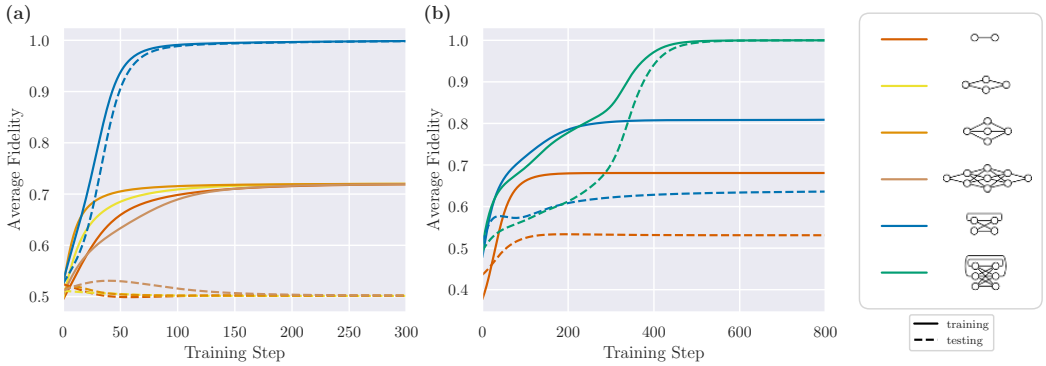


Figure 4.4: **Learning the delay-by-1 and -2 channel with pure states and the local cost.** Panel (a) shows the average fidelity on the training (drawn-through) and the validation (dashed) set for each training step for ff DQNNs with different architectures (yellow to red tones) and a simple DQRNN (blue) with 1 memory qubit learning the delay-by-1 channel. Panel (b) shows the average fidelity on the training (drawn-through) and the validation (dashed) set for each training step for a simple ff DQNN (red), a simple DQRNN with 1 memory qubit (blue), and a simple DQRNN with 2 memory qubits (green) learning the delay-by-2 channel. In both panels, we used $N = 20$ training pairs and a learning rate of $\eta = 0.06$.

memory of k qubits should suffice as the DQRNN can then simply swap the inputs in the memory and output them at a later point. It would essentially save the last inputs in the memory. Less than k qubits should not be enough to learn the delay-by-k channel well, as we cannot save the last k inputs in the memory.

As a proof-of-concept, we look at the examples of $k = 1, 2$ and check how well ff DQNNs, and DQRNNs learn the delay channel using mainly the (validation) cost as the figure of merit. The training and validation costs for each training step are shown in Figure 4.4.

For $k = 1$, Panel (a) shows the average fidelity for each training step for four different ff DQNN architectures and a simple DQRNN with 1 memory qubit and no hidden layers. We can see that the DQRNN learns the delay-by-1 channel well, i.e., the cost on the training set reaches nearly 1, and also generalizes well to the validation set, as the cost on the validation set reaches nearly 1. While the ff DQNNs seem to learn a bit, in particular, the cost on the training set rises from 0.5 to 0.7, they do not generalize at all, since the cost on the validation set does not (except for a small bump) rise at all. Independent of the architecture, even at the end of the training, the average fidelity on the validation set is 0.5, which is the average fidelity between two pure states chosen randomly w.r.t. the Haar measure. That means that the ff DQNN at most learned some of the training samples by heart and overfits. This is expected since there is no direct relation between the x^{th} input and x^{th} output of the delay channel.

For $k = 2$, Panel (b) shows the average fidelity for each training step for a ff DQNN, a DQRNN with 1 memory qubit, and a DQRNN with 2 memory qubits. None of the architectures have hidden layers. As expected, the DQRNN with 2 memory qubits learns and generalizes well, the DQRNN with 1 memory qubit learns and generalizes less well, and

Figure 4.5: **Generalization behavior for the delay-by-1 channel.** We train two ff DQNNs (red and orange) and a DQRNN without hidden layers on the delay-by-1 channel with $N = 1, ..., 10$ training pairs and learning rate $\eta = 0.06$. The average fidelity is then evaluated on the training set and on a validation set of size 30 (stars). This process is repeated 50 times. The plot shows the average fidelity for each number of training steps.

the ff DQNN learns a map fitting the training data even worse and does not generalize. The DQRNN with 2 memory qubits has the same number of memory qubits as the delay-by-2 channel, which swaps the input qubits into the memory space and stores them there. So, in principle, the DQRNN with 2 memory qubits can learn to do the same, essentially saving the last 2 inputs in the memory, which is not at all possible for the ff DQNN. The DQRNN with 1 memory qubit can only store part of that information in the memory.

Additionally, we studied the generalization behavior for $k = 1$ with two ff DQNNs and a DQRNN without hidden layers. To do so, we repeat the following process 50 times for each number of training pairs $N = 1, ..., 10$: We generate a training set of size N and a validation set of size 30. Then, we train the DQNNs with the learning rate $\eta = 0.06$ for 1000 steps on the training set. Subsequently, we evaluate the average fidelity on both the training set and the validation set. Figure 4.5 shows the average over the evaluated fidelities for each number of training pairs N. One can see that, independent of the number of training pairs, the DQRNN learns almost perfectly if we only take the training cost into account. The average fidelity on the validation set rises with the number of training pairs up until $N = 8$, where it is close to perfect. Hence, with a DQRNN with 1 memory qubit, only 8 training pairs are required to learn the delay-by-1 channel and generalize to unseen data. On the other hand, if we only look at the training cost, independent of the used architecture, both ff DQNNs can still predict the output for $N = 1$. Their performance on the training set drops with each added training pair approaching an average fidelity of approximately 0.7 because each added training pair makes it harder to find a relationship between the seemingly random inputs and outputs the DQNNs are trained on. For $N = 1$, the ff DQNN can still easily find a unitary mapping the input to the output of the training set, which gets harder with every seemingly random input-output pair. The ff DQNNs do not generalize at all. The average fidelity on the validation set stays at 0.5, which is the average fidelity between two states chosen randomly w.r.t. the Haar measure. We again see that the ff DQNNs can only learn the training examples by heart and overfit.

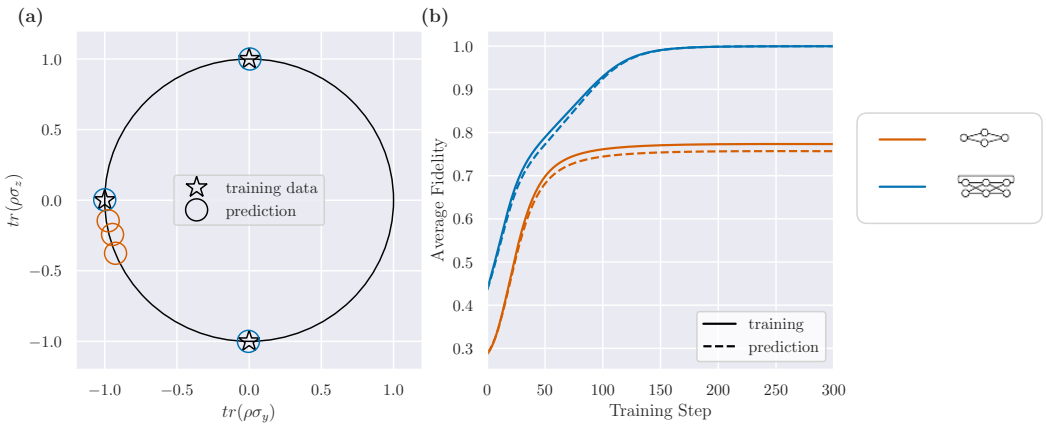


Figure 4.6: **Time evolution of $|0\rangle$ w.r.t. a time-dependent Hamiltonian and $\tau = 0.5$.** Panel (a) shows the (y, z)-plane of the Bloch sphere. Stars mark the time evolution of the state $|0\rangle$ under the Hamiltonian in Equation (4.6.1) with $\tau = 0.5$, i.e., we show $|\psi(0)\rangle = |0\rangle, |\psi(\tau)\rangle = |-i\rangle, |\psi(2\tau)\rangle = |1\rangle, |\psi(3\tau)\rangle = |-i\rangle, |\psi(4\tau)\rangle = |0\rangle$, and so on. We use $N = 10$ training points produced by this time evolution to train a ff DQNN (red) and a DQRNN with 1 memory qubit (blue) using the learning rate $\eta = 0.05$ for $T = 1000$ training steps. The DQNNs are then used to predict the time evolution for the next 20 time points. These predictions are shown in Panel (a) with colored circles. Panel (b) shows the average fidelity for each training step for both the training set (drawn-through) and the prediction (dashed).

As expected, for the studied examples of the delay-by-k channel with $k = 1, 2$, the DQRNNs with k memory qubits generalize best to unseen data. In contrast, the ff DQNNs do not generalize at all, independent of the underlying architecture. The performance of the DQRNN with one memory qubit performs somewhat in the middle for $k = 2$. With $k = 1$, the DQRNN with one memory qubit only requires $N = 8$ training pairs to generalize nearly perfectly.

Time evolution

The other task we consider is the time evolution of a state governed by a time-dependent Hamiltonian. It is defined as follows: we know the time evolution of a pure state $|\psi(t)\rangle$ in a given time interval $[T_1, T_2]$ on N equidistant points $t = T_1, T_1 + \tau, ..., T_2$ where $\tau = \frac{T_2 - T_1}{N}$. Our task is then to predict the time evolution on the interval $(T_2, T_3]$, where $T_3 = T_2 + \tilde{N}\tau$ for some $\tilde{N} \in \mathbb{N}$.

We choose $\mathcal{H} = \mathbb{C}^2$, the Hamiltonian

$$H(t) = \frac{\pi}{2} f(t)(1 - \sigma^x), \quad f(t) = \begin{cases} 1, & t \in [2k, 2k+1) \\ -1, & t \in [2k+1, 2k) \end{cases}, \quad k \in \mathbb{N}, \tag{4.6.1}$$

and $|\psi(0)\rangle = |0\rangle$. Under time-evolution, this leads basically to a rotation from $|0\rangle$ to $|1\rangle$ and back in the (y, z)-plane of the Bloch sphere. It is $\mathrm{tr}(\rho\sigma_x) = 0$, $\mathrm{tr}(\rho\sigma_y) \leq 0$, and $\mathrm{tr}(\rho\sigma_z)$ oscillates between -1 and 1.

For $\tau = 0.5$, the state starts in $|0\rangle$, then evolves to $|-i\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{i}{\sqrt{2}}|1\rangle$, to $|1\rangle$, to $|-i\rangle$, to

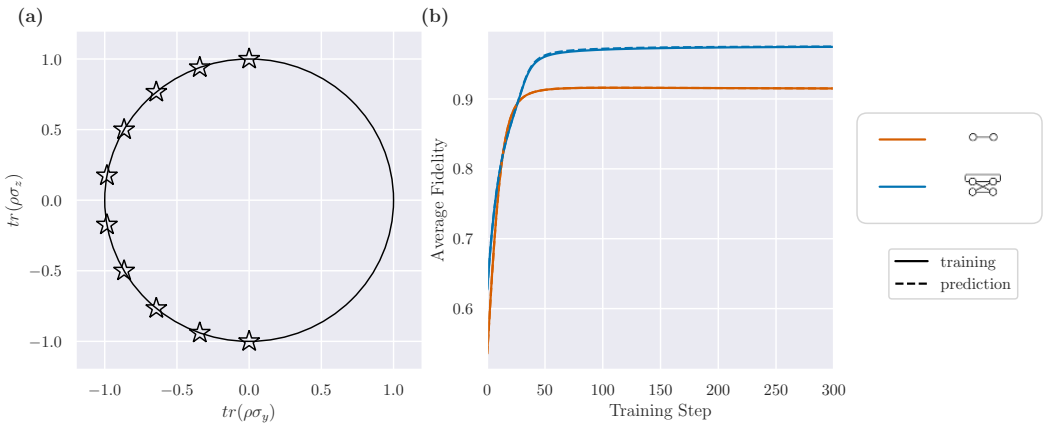


Figure 4.7: **Time evolution of $|0\rangle$ w.r.t. a time-dependent Hamiltonian and $\tau = 0.1$.** Panel (a) shows the (y, z)-plane of the Bloch sphere. Stars mark the time evolution of the state $|0\rangle$ under the Hamiltonian in Equation (4.6.1) with $\tau = 0.1$, i.e., we show $|\psi(0)\rangle, |\psi(\tau)\rangle, ... |\psi(N\tau)\rangle$. We use $N = 100$ training points produced by this time evolution to train a ff DQNN (red) and a DQRNN with 1 memory qubit (blue) using the learning rate $\eta = 0.06$ for $T = 2500$ training steps. The DQNNs are then used to predict the time evolution for the next 200 time points. Panel (b) shows the average fidelity for each training step for both the training set (drawn-through) and the prediction (dashed).

$|0\rangle$, to $|-i\rangle$, and so on. This time evolution is marked with stars in Panel (a) of Figure 4.6, which shows the (y, z)-plane of the Bloch sphere. We use $N = 10$ training points produced by this time evolution to train a ff DQNN (red) and a DQRNN with 1 memory qubit (blue) using the learning rate $\eta = 0.05$ for 1000 training steps. The DQNNs are then used to predict the time evolution for the next 20 time points. These predictions are shown in Panel (a) with colored circles. Panel (b) of the same figure shows the average fidelity for each training step for both the training set (drawn-through) and the prediction (dashed). We can see in Panel (b) that the average fidelity for the DQRNN reaches higher values near 1, both in training and prediction, than the ff DQNN even if the difference is not as pronounced as for the delay channel. The predicted states in Panel (a) for the ff DQNN are all very close to $|-i\rangle$, and the time-evolution is not predicted at all. This is probably due to $|-i\rangle$ appearing most often in the training set and the ff DQNN not being able to predict in which direction to turn from there. On the other hand, the DQRNN is also able to store the last visited state and get some kind of direction from that. We can see that the predicted states by the DQRNN are nearly on top of the actual time evolution. In total, the DQRNN performs much better for this task.

For $\tau = 0.1$, we visit nine states in between $|0\rangle$ and $|1\rangle$. The visited states are again marked with stars in Panel (a) of Figure 4.7, which shows the (y, z)-plane of the Bloch sphere. We use $N = 100$ training points produced by this time evolution to train a ff DQNN (red) and a DQRNN with 1 memory qubit (blue) using the learning rate $\eta = 0.06$ for 2500 training steps. The DQNNs are then used to predict the time evolution for the next 200 time points. Panel (b) of the same figure shows the average fidelity for each training step for both the training set (drawn-through) and the prediction (dashed). Compared to the case $\tau = 0.5$, with $\tau = 0.1$ the difference in performance of the DQRNN and the ff DQNN is not as pronounced, and the performance of the DQRNN is not optimal. However, the DQRNN

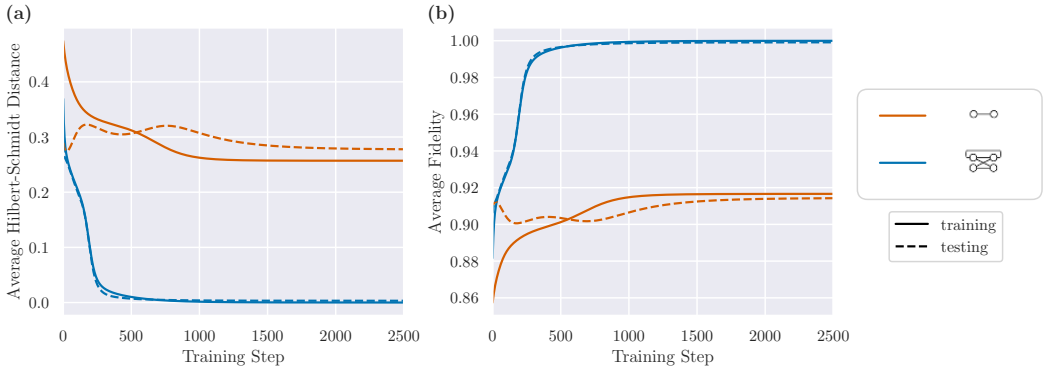


Figure 4.8: **Learning the delay-by-1 channel with mixed states and the local cost.** The plots show the average Hilbert-Schmidt distance (Panel (a)) and fidelity (Panel (b)) on the training set (drawn-through) and the validation set (dashed) for each training step. We used $\eta = 0.05$ as the learning rate and $N = 20$ training pairs.

still performs better than the ff DQNN and reaches fidelities of approximately 0.931.

For both values of τ, the DQRNN performs better than the ff DQNN. The difference in performance is not as pronounced as for the delay channel, especially for $\tau = 0.1$. Note that there is virtually no difference in performance on the interval $[T_1, T_2]$ and the interval $(T_2, T_3]$. For this periodic time evolution, the DQNNs can hence learn the time evolution on an interval as well as they can predict the time evolution on the next interval.

4.6.2 Local Cost and Product Data with Mixed Target Output

In this section, we focus on mixed target outputs. As a proof of concept, we use the same example of the delay-by-1 channel discussed before with mixed outputs. The results are shown in Figure 4.8. We show both the average Hilbert-Schmidt distance (Panel(a)) on which the DQNNs were trained and the more physically relevant fidelity (Panel(b)). Panel (b) looks similar to the plot with pure target outputs, only that the starting fidelity and fidelity reached by the ff DQNN are higher as the average fidelity of two random mixed states chosen via purification and the Haar measure is higher than the one for pure states.

4.6.3 Global Cost and Product Data with Pure Target Output

In contrast, when we consider the global cost with pure target outputs, we see that the cost is generally much smaller than the local cost as it translates to the product instead of the average of the individual fidelities. For $N = 20$, it has values around 10^{-7} for randomly chosen states. We again use the delay-by-1 channel discussed before for a proof-of-concept. The results are shown in Figure 4.9. Panel (a) again shows the fidelity for each training step. As the fidelity at the start and for the ff DQNN is so low that we cannot see any training process for the ff DQNN, we show its logarithm in Panel (b). Other than the lower starting cost, the results look similar to before.

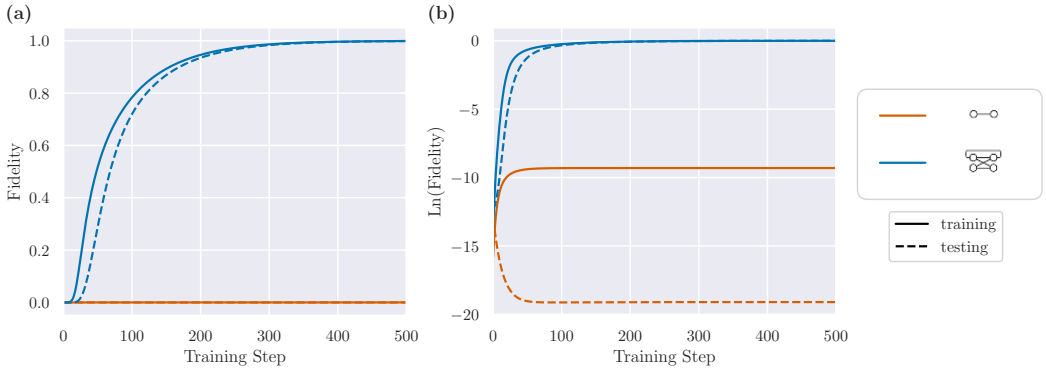


Figure 4.9: **Learning the delay-by-1 channel with pure states and the global cost.** The plots show the fidelity (Panel (a)) and its logarithm (Panel (b)) on the training set (drawn-through) and the validation set (dashed) for each training step. We used $\eta = 0.005$ as the learning rate and $N = 20$ training pairs.

4.7 Implementation on NISQ Devices

Implementing the DQRNNs on NISQ devices works analogously to the realization of ff DQNNs described in [250]. To implement the underlying DQNN of the DQRNN we use N_p parametrized unitaries. The parameter vector $\vec{p} \in \mathbb{R}^{N_p}$ is updated using gradient descent, i.e., $\vec{p} \mapsto \vec{p} + \mathrm{d}\vec{p}$ where $\mathrm{d}\vec{p} = -\eta \nabla C\left(\vec{p_t}\right)$ and η is the learning rate. The gradient is estimated numerically, i.e.,

$$\left(\nabla C\left(\vec{p_t}\right)\right)_k \approx \frac{C\left(\vec{p_t} + \epsilon \vec{e_k}\right) - C\left(\vec{p_t} - \epsilon \vec{e_k}\right)}{2\epsilon}$$

with $\left(\vec{e_k}\right)^j = \delta_k^j$, $k, j = 1, ..., N_p$, $\epsilon > 0$. If the target output is pure, the cost can be estimated with the SWAP trick as described in [82, 245].

As already discussed, we can either use the global or local cost. The global cost has the advantage that we can also learn correlations between the outputs, so after training, the DQRNN can be completely used on different states, and the output is either correctly entangled or correctly separable. In the following, we assume that qudits can be reset and used again.

When employing the global cost, we first evaluate the full DQRNN and then use the SWAP trick on the global output and target output. This requires at least $2Nm_{\text{out}}$ qudits for N iterations.

When estimating the local cost at the x^{th} output, we use the underlying DQNN x times on the local input and memory. The local output of the first $x-1$ iterations and the local memory of the x^{th} iteration can be ignored or reset, and the SWAP test only has to be performed on at most $2\|m\|_\infty$ qudits. For this, x iterations of the DQNN are required. The process then has to be repeated for $x = 1, \ldots, N$. In total, we have to iterate over the underlying DQNN at least $N(N+1)/2$ times for each data point estimating the cost and necessitate not more than $2\|m\|_\infty$ qudits. This makes the local cost easier to estimate on NISQ devices.

4.8 Conclusion and Outlook

In conclusion, we presented a fully quantum RNN architecture capable of learning general causal quantum automata. As our architecture is based on dissipative quantum neural networks (DQNNs), we name it dissipative quantum recurrent neural network (DQRNN). We presented different classical training algorithms for both product and entangled data. The size of the needed matrix multiplication only scales with the width of the network and the bond dimension of the input and target output of the training data, but not the depth of the network. The presented examples, the delay channel and the time evolution of a state under a time-dependent Hamiltonian, show that the DQRNN is able to deal with this kind of sequential data better than ff DQNNs. For the delay channel, the DQRNN is able to generalize from a few training points.

Several promising applications remain to be explored, particularly in the study of entangling channels with memory, the generation of entangled states of varying lengths, and potential applications in solid-state physics. Moreover, it is important to delineate the specific problems for which quantum memory offers advantages over classical memory. Future work could also investigate the relationship between this method and TN approaches for training classical NNs, as outlined in [69]. A common strategy for handling sequential data in ff NNs is to incorporate a *history*, where not only the input at time t is used but also inputs from previous time steps $t-n, ..., t$, thereby increasing the dimensionality of the input space. As discussed in Section 3.11.3, DQNNs encounter the issue of barren plateaus, where the cost landscape becomes exponentially flatter as system size increases. A potential avenue for future research is to compare the barren plateaus encountered in lower-dimensional DQRNNs with those arising from history states in DQNNs. Additionally, when studying the global cost, we noticed that it can help with performance to scale the learning rate with the inverse of the fidelity. Further developments in this area could involve studying if this can help more generally with the trainability issues of (shallow, sparse) DQNNs with a global cost.

Data availability The python code is available at `https://github.com/qigitphannover/DeepQuantumNeuralNetworks/tree/master/QRNN`.

5

Reinforcement Learning for Fiber Coupling

This chapter is based on the work [105].

Current candidates for implementing qubits include trapped ion qubits, superconducting qubits, photonic qubits, and neutral atom qubits. Many of those approaches use complicated laser systems in their realization [91]. In those laser systems, many control and alignment tasks must be repeated frequently, either by hand or automatically. Here, aligning an experiment means steering a laser beam correctly through an optical experiment. Control, most of the time, refers to keeping a dynamic experiment at a given fixed point. In many labs using lasers [299, 300], including quantum information labs [91–94], optical fibers, consisting of glass or plastic, are used for the transmission of light [301]. Fiber coupling, i.e., guiding a laser beam into the fiber, is employed to transition from free-space optics to fiber optics.

The ML technique natively most suited for control tasks is RL [43]. In the field of optics, RL has mainly been employed in adaptive optics [302–307], optical networks [308–316], thin films, optical nanostructures, and optical layers [317–320]. It has been less used in optical table-top experiments, on which we focus here. Like in the general control field [95–99], RL has mostly been applied in simulated environments. Either the training and testing were performed in simulation [321–324], or the training was performed in simulation, and the trained agents were tested in an experiment [325–329]. Training in an experiment is very rare [99]. Examples include combining two pulsed laser beams [330] and generating a white light continuum [331]. Applying RL to real-world scenarios comes with several challenges, such as time-consuming training, partial observability, and noise [97, 98]. When overcoming these challenges, RL could help simplify lab work. Hence, in this chapter, we discuss employing RL for fiber coupling, which is a part of many optical experiments.

Using RL for fiber coupling, we had to overcome three main challenges: imprecise actuators, partial observability, and time-consuming training. In our initial experiment, we used PPO and the typical reset method of choosing random actuator positions in a given interval. Even after two weeks of continuous runtime in the laboratory, the return stayed at its minimal value because light exiting the fiber was rarely observed by the agent. This was mainly due to the speed and inaccuracies of the simple open-loop stepper motors employed in the experiment. We thus had to change our reset method to not fully rely on absolute actuator positions. This change in the reset method led us to always start our episodes with actuator positions at which a fraction of power behind the fiber could be observed. Additionally,

the imprecise actuators lead to our actions being very noisy. Note that this noise is not artificially introduced in the actions for exploration, as done, e.g., in [332–336], but the noise is inherent to the environment. We deal with this by directly training our agent on the experiment, so we do not need an accurate noise model. Hence, using a relatively sample-efficient algorithm is critical. As discussed in Section 2.2.2, PPO does not use a replay buffer, which makes it less sample-efficient than value-based methods. Those, in turn, historically were mostly used on discrete action spaces. In the last few years, more sample-efficient algorithms for continuous action spaces like DDPG, TD3, SAC, and TQC were developed. This makes one main problem – the time-consuming training – of using RL in real-world scenarios less severe. Another important modification was the design of the environment, especially of the observations and rewards. To save time, we used a virtual testbed for tuning the environment so that the agent needs fewer interactions with it and can better deal with the partial observability of its environment.

Despite the challenges, particularly the noisy actions, we demonstrate how an RL agent successfully learns to steer two mirrors with two axes each to couple light into a fiber. Our main figure of merit is the fiber coupling efficiency, which is the power of the outcoming laser beam divided by the power of the incoming beam. The agent reliably reaches its goal of $90\% \pm 2\%$ efficiency, which is comparable to human experts, starting from a low coupling efficiency (mostly between 20% and 40%). For this, it needs around 4 days of training time on the experiment and no pre-training on the virtual testbed. If the goal is lower, e.g., $87\% \pm 2\%$, an agent can be trained in less than a day. These training times were made possible by careful tuning of the environment and algorithm selection in the virtual testbed. Using these trained RL agents in the lab, given some starting efficiency, fiber coupling can be performed remotely and automatically, saving the human experimenter time.

In Section 5.1, we describe the task of fiber coupling from an RL perspective. Afterward, the design of our environment is presented in Section 5.2, which we tune using a virtual testbed in Section 5.3. In Section 5.4, we discuss results from the lab.

5.1 Fiber Coupling

We will restrict our discussion of fiber coupling to the aspects important for the environment design and leave out the experimental design aspects, which can be found in [105]. The experimental setup is depicted in Figure 5.1. The goal of fiber coupling is to guide an incoming laser beam into an optical fiber so that either a given percentage of its power or as much light as possible is observed exiting the fiber. The fiber's output power is influenced by the beam's shape and waist size and the position and angle at which the beam enters the fiber [337]. The beam's shape and waist size cannot be changed by the agent in our setup. As light has to impinge both at a specific position and with a specific angle to efficiently couple to the fiber [301, 337], we need at least two mirrors, each tiltable in both the horizontal (x) and vertical (y) axis, for fiber coupling. The setup includes two mirrors with two motorized axes each for automated fiber coupling and two hand steering mirrors for fiber coupling by a human experimenter.

The only sensors we employ are two power meters. One is used to estimate the input power P_{in} of the setup, and the other measures the output power P_{out}. We can then calculate the *coupling efficiency* or *(normalized output)* power $P = \frac{P_{\text{in}}}{P_{\text{out}}}$. Without other sensors that would complicate the setup, our environment is partially observable.

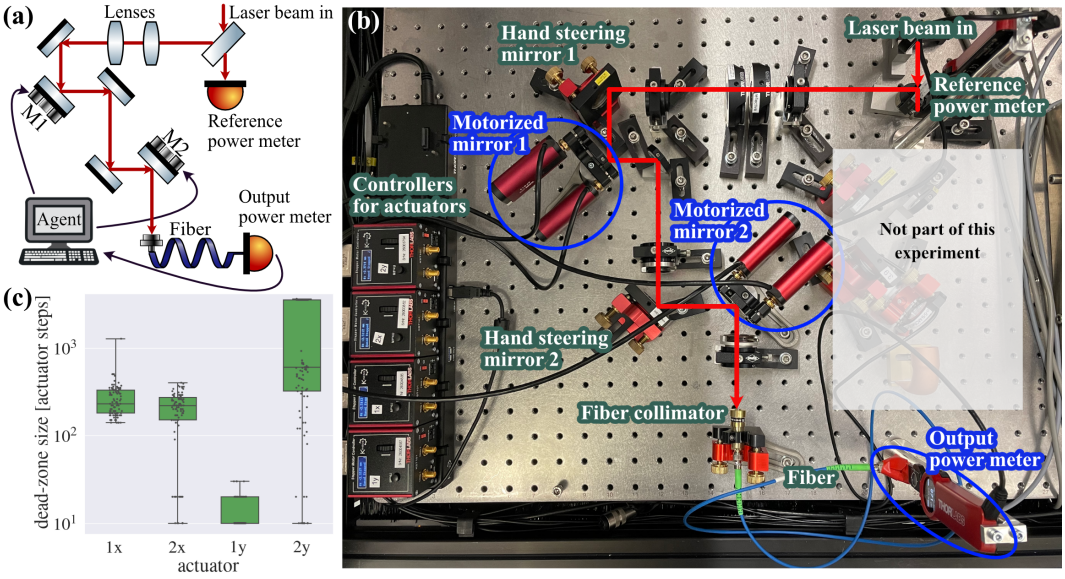


Figure 5.1: **Fiber Coupling: Setup and Dead-Zone Characterization [105].** Panel (a) shows the schematic setup of the experiment. Note that the lenses can be used to manually adjust the beam size to fit the mode of the fiber coupler. Panel (b) shows a picture from the lab with the laser beam drawn in. Panel (c) shows the dead-zone size for the four actuators.

Additionally, the motors used are inaccurate. They exhibit *dead-zones*, which arise from mechanical hysteresis of the motor gearbox. Within the dead-zone, the mirror tilt angle is unchanged while the motor rotates until it overcomes the hysteresis. The size of the dead-zone is dependent on the movement history and is maximized by large movements in the opposite direction before changing direction. The actual amount of steps needed to overcome the dead-zone is stochastic, with samples depicted in Figure 5.1 (c). More information on the reasons for this can be found in [105]. This leads to our actions given by the movement of the actuators being very noisy and us not being able to fully depend on absolute motor positions for our reset method.

5.2 Environment Design

Our Environment can be seen as an unknown episodic POMDP. Sampling from it, we get a stochastic process of observations, actions, and rewards $O_0, A_0, R_1, O_1, ..., A_\tau, R_\tau, O_\tau$ up to the end of the episode at time $t = \tau \leq T$, where T is the maximum episode length. We now describe the definition of our POMDP by specifying our actions, observations, rewards, and episodes and resets, i.e., specifying when an episode ends and what happens afterward. The default parameters for the number of parameters used are shown in Tables 5.1 and 5.3. The environments - both the experimental and the virtual environment - are implemented using Gymnasium [338].

5.2.1 Action

Like in most control environments [43], the action corresponds to moving the actuators, i.e., tilting the mirrors in our case. The agent controls four actuators, and thus, the action space is four-dimensional. As we want it to be able to do larger movements at the start of each episode and smaller movements for fine-tuning, we choose a continuous action space. As ML algorithms usually deal better with values on the same scale [339], we choose our action space to be $[-1, 1]^{\times 4}$. Each action $A_t = (A_{\mathrm{m1}x}, A_{\mathrm{m1}y}, A_{\mathrm{m2}x}, A_{\mathrm{m2}y})$ is then multiplied with the maximal action $a_{\max}$, rounded to the next integer, and send to the controller. For example, $A_{\mathrm{m1}x}$ belongs to the actuator tilting mirror 1 in the horizontal (x) direction. The exhibited dead-zone of the actuators and other noise sources lead to the actions being very imprecise or noisy in the experimental environment.

5.2.2 Observation

We do not have any access to the exact mirror positions and beam angles. Even if we had, we would not want the absolute positions to be part of our observation, as this would lead to the agent learning to move to specific optimal positions. If the experiment is then aligned differently at some earlier point of the experiment, the agent would still move to the earlier optimal positions, which would not be optimal anymore (see Section 5.2.2). Hence, the agent would not be able to find the optimum again and would be useless in the very situations in which we want to use it. Therefore, we avoid using the absolute actuator positions as part of the observation.

The power exiting the fiber P_{out} and a portion of the light entering the experiment P_{in} are measured much more frequently than the agent performs actions. Using these measurement results, we can get the coupling efficiency or normalized power behind the fiber $P_t = \frac{P_{\mathrm{out},t}}{P_{\mathrm{in},t}}$ at nearly all times $t \in \mathbb{R}$. However, many different positions can lead to the same power output. This makes our environment partially observable and underdetermined. For this kind of environment, people usually use a history of observed quantities after each action as observation (see e.g. [325,330]). This would leave us with the observation $O_t = (P_{t-n}, ..., P_{t-1}, P_t)$ at time $t \in \mathbb{N}$, where $n \in \mathbb{N}$ is the history length. As already mentioned, in addition to observing the normalized power between actions, we can also observe it while performing the actions. During each action A_t, we can record a list of powers $(P_t, P_{t+1/m_t}, ..., P_{t+1})$. The number of measurements m_t performed while executing the action A_t heavily depends on the size of the action, i.e., m_t is not the same in each environment step. That makes it hard to use the full list of powers as observation, as classical ff NNs, which are used in modern RL algorithms by default, need their input to be of the same size. Instead, we use the average power

$$P_{\mathrm{ave},t} = \frac{1}{m_t + 1} \sum_{i=0}^{m_t} P_{t-1+i/m_t},$$

the maximum power

$$P_{\max,t} = \max_{i=0,...,m_t} P_{t-1+i/m_t} = P_{t-1+x_{\max}},$$

and its relative position

$$x_{\max,t} = \frac{1}{m_t + 1} \arg \max_{i=0,...,m_t} P_{t-1+i/m_t}$$

Table 5.1: **Environment parameters.** The table shows the default environment parameters for experiments. The parameter P_{min} corresponds to the power over which we want to start our experiment (realistically, this is closer to $P_{\text{min}} - 0.1$ in the experiment), P_{fail} is the power at which the agent fails, P_{goal} is the power at which the agent reaches its goal, a_{max} is the maximum action, T is the maximum episode length, n is the history length. At the start of each reset, we move to the neutral positions if the power is smaller than $P_{\text{neutral},1}$ or if the reset number is divisible by l. During the resets, we move to the neutral positions if the power is smaller than $P_{\text{neutral},2}$.

P_{min}	P_{fail}	P_{goal}	$P_{\text{neutral},1}$	$P_{\text{neutral},2}$	a_{max}	T	n	l
0.2	0.05	$[0.75, 0.91]$	0.09	0.04	$6 \cdot 10^3$	30	4	10

to be part of the observation. In addition, we include the actions in the observation so that the RL agent still remembers what it had to do to get the observed powers. This leaves us with the observation

$$O_t = \left((P_{k-1}, A_{k-1}, P_{\text{ave},k}, P_{\text{max},k}, x_{\text{max},k})_{k=t-n,\dots,t}, P_t \right).$$

This means the agent observes the power before taking an action, the action it took, the average power, maximum power and its relative position while taking an action, and the power after taking an action[1] in each of the last n steps.

5.2.3 Episodes and Resets

We define our environment as an episodic POMDP, i.e., we repeat the following in a loop: First, the environment is reset to a certain distribution over starting states. Then, the agent performs actions, leading to changes in the environment and rewards until the episode ends. We then reset again, and the cycle starts anew.

The end of an episode is usually reached when a certain termination condition is met. This can, e.g., be reaching a terminal state or a maximum number of environment steps. In our environment, we employ the following termination conditions: Firstly, the episode ends after $t = T \in \mathbb{N}$ environment steps. This is needed so that the episodes cannot go on forever, the agent does not get caught up in certain situations, and the return between episodes is comparable. Secondly, we implement a failing condition, i.e., the episode ends if $P_t < P_{\text{fail}}$. This termination condition is required for practical reasons in the experimental environment, in particular, to not lose track of the power because the signal becomes indistinguishable from the noise. As the absolute actuator positions are not reliable, they cannot be used to reset the environment. Hence, we have to rely on relative positions for the reset, which would easily get intractable if we perform too many environment steps outside of any feedback. Instead of complicating the setup with additional sensors, we decided to end the episode (with a negative reward) if the power drops below a failing power P_{fail}. Thirdly, we set a *goal*, i.e., the episode ends (with a positive reward) if $P_t > P_{\text{goal}}$. This last termination condition is more of a design choice and less necessary. In most experimental settings, it is enough for us to exceed a certain power. After reaching that power, it is less important to get the maximum possible power and more important to have a constant power output. Thus, we set a goal power P_{goal}. However, on the virtual testbed, we also analyze what

[1] This is the same as the power before taking the next action, which is why it does not appear explicitly in the history part of the observation.

happens if we do not set a goal and the agent's aim is to optimize the power. Those results can be found in Appendix B.4.

After an episode ends, we reset the environment. It is common to reset all actuators to a given parameter range but that does not work in the experimental environment due to the motors' imprecision. Instead, we perform the procedure detailed in Algorithm 13, where get_power() gets the normalized power, move_by(v) moves the actuators by v, and move_to(x) moves the actuators to position x. The so-called *neutral positions* x_{neutral} are actuator positions at which the power is high at the start of training, which in later stages is not guaranteed anymore due to the motors' inaccuracies. The reset method is a combination of moving to the neutral positions, doing random steps while the power is high ($P > P_{\text{lim}}$, where P_{lim} is a random power in the interval $[P_{\text{min}} + 0.1, P_{\text{goal}}]$), and moving in a direction in which the power increases while the power is low ($P < P_{\text{min}}$). This procedure makes the episodes not fully independent of each other. However, this generally has only a small impact that is even smaller for small l; see Section 5.3.6. Due to the dead-zone of the actuators, full independence would not have been possible in the experimental environment.

5.2.4 Reward

Let us now design the reward. In each step, our agent can reach a goal, fail, or none of these. For each of these cases, we have to define a reward which we call r_{goal}, r_{fail} and r_{step}, respectively. A common choice in modern RL environments for classic games like chess or Go is to choose $r_{\text{goal}} = +1$, $r_{\text{fail}} = -1$ or $r_{\text{fail}} = 0$, and $r_{\text{step}} = 0$ [44, 45, 147]. In those games, it usually does not matter by how much the agent wins and how it performs while playing. This kind of sparse reward is usually more accurate, in the sense that no false bias is introduced, than dense rewards. However, sparse rewards can lead to longer training times, as dense rewards provide the agent with more feedback in the early stages of training [47]. Because we want to train on an experiment, sample efficiency is of utmost importance. Thus, we define intermediate rewards and choose $r_{\text{step}} \neq 0$.

We start with designing the reward the agent gets in every step. The reward should be higher for higher power values. Additionally, it should be rewarding for the agent to go from a small power, e.g., 0.2, to a higher power, e.g., 0.5, but it should still be rewarding to go from high powers, e.g., 0.85, to slightly higher powers, e.g., 0.86. This means that both, $r(P = 0.5) - r(P = 0.2)$ and $r(P = 0.86) - r(P = 0.85)$, should not be too small. Hence, we combine an exponential and a linear function and get

$$r_{\text{step}} = \frac{A_s}{T} \left((1 - \alpha_s) \exp\left(\beta_s (P_t - P_{\text{goal}})\right) + \alpha_s \left(P_t - P_{\text{min}}\right) \right).$$

We can use $\alpha_s \in (0, 1)$ to tweak the importance of the agent noticing a difference in high powers vs. low powers. The pre-factor $A_s \in \mathbb{R}_{>0}$ can be used to adjust the impact of the step reward in comparison to the fail and goal reward. We divide it by the maximum episode length T to make this comparison easier when summing up r_{step} over the full episode and to not need to adjust the pre-factors when we change the episode length. The factor in the exponent $\beta_s \in \mathbb{R}_{>0}$ can be used to tweak the slope of the exponential function.

Now, let us design the reward in the case of failure. The reward for failure should be negative and smaller than the lowest step reward. To us, it matters how fast the agent is failing and

Algorithm 13 Reset method for fiber coupling environment

$P \leftarrow P_t, v \leftarrow -\frac{A_t}{\|A_t\|}, p \leftarrow$reset number ▷ *Initialize power P as the last power observed, direction v as normalized negative last action and p as the reset number*
if $P < P_{\text{fail}}$ **then**
    move_by$(-A_t a_{\max})$ ▷ *Reverse the last action in case this leads to an increase in power*
    $P \leftarrow$ get_power() ▷ *get_power measures the normalized power*
if $p \mod l = 0$ or $P_t < P_{\text{neutral},1}$ **then** ▷ *Every $l \in \mathbb{N}$ resets or if power still small*
    move_to(x_{neutral}) ▷ *Move actuators to neutral positions $x_{neutral}$*
    $P \leftarrow$ get_power()
    if $P_t < P_{\text{neutral},2}$ **then** ▷ *If it is even smaller, do random steps*
        Choose $w \in [-10^4, 10^4]^{\times 4}$ at random (uniform distribution)
        move_by(w)
        $P \leftarrow$ get_power()
if $P_t \geq P_{\min}$ **then** ▷ *If the power is relatively high*
    Choose $P_{\text{lim}} \in [P_{\min} + 0.1, P_{\text{goal}}]$ at random (uniform distribution)
    while $P > P_{\text{lim}}$ **do** ▷ *While $P > P_{lim}$, do random steps with the actuators*
        Choose $w \in [-10^4, 10^4]^{\times 4}$ at random (uniform distribution)
        move_by(w)
        $P \leftarrow$ get_power()
while $P < P_{\min}$ **do** ▷ *If the power is relatively small*
    if $P_t < P_{\text{neutral},2}$ **then** ▷ *If power is very small, move to neutral positions*
        move_to(x_{neutral})
        $P \leftarrow$ get_power()
        if $P_t < P_{\text{neutral},2}$ **then** ▷ *If it is still small, do random steps*
            Choose $w \in [-10^4, 10^4]^{\times 4}$ at random (uniform distribution)
            move_by(w)
            $P \leftarrow$ get_power()
    else ▷ *If it is not that small, but still small, move actuators as long as power increases*
        for $i \in$ random_permutation$(1, 2, 3, 4)$ **do** ▷ *Move the actuators in a random order*
            $\tilde{P} \leftarrow P$
            Choose $w \in [5 \cdot 10^3, 2 \cdot 10^4]$ at random (uniform distribution)
            move_i_by(wv_i) ▷ *Move current actuator a random step in current direction*
            $P \leftarrow$ get_power()
            while $\tilde{P} - P \geq 0$ **do** ▷ *If power increased*
                $\tilde{P} \leftarrow P$
                Choose $w \in [5 \cdot 10^3, 2 \cdot 10^4]$ at random (uniform distribution)
                move_i_by(wv_i) ▷ *Move current actuator a random step in current direction*
                $P \leftarrow$ get_power()
            if $\tilde{P} - P < -0.002$ **then** ▷ *If power decreases significantly, reverse last steps*
                move_i_by$(-wv_i)$
                $P \leftarrow$ get_power()
        $v \leftarrow -v$ ▷ *Reverse movement direction*
Choose $w \in [-10^3, 10^3]^{\times 4}$ at random (uniform distribution)
move_by(w)
$P \leftarrow$ get_power()
$p \leftarrow p + 1$

Table 5.2: **Reward hyperparameters.** The table shows the default reward hyperparameters for experiments.

Parameter	A_s	A_f	A_g	α_s	α_f	α_g	β_s	β_{f1}	β_{f2}	β_{g1}	β_{g2}
Value	10	100	100	0.9	0.5	0.5	5	5	5	5	1

by how much. Hence, we define

$$r_{\text{fail}} = -A_f \left((1-\alpha_f) \exp\left(-\beta_{f,1} \frac{t}{T}\right) + \alpha_f \exp\left(-\beta_{f,2} \frac{P_t}{P_{\text{fail}}}\right) \right).$$

In this way, the agent is punished less the later it fails and the higher the power is with which it fails. Again, we can use $\alpha_f \in (0,1)$ to weigh the importance of both of these factors, $\beta_{f,1}, \beta_{f,1} \in \mathbb{R}_{>0}$ to determine the reward functions' curvature, and $A_f \in \mathbb{R}_{>0}$ to weigh the fail reward in comparison to the other two.

Lastly, we design the reward in the case of the agent reaching the goal. This reward should be positive and higher than the maximum step reward. To us, it matters how fast the agent is reaching the goal and how high the power is that it reaches. Hence, we define

$$r_{\text{goal}} = A_g \left((1-\alpha_g) \exp\left(-\beta_{g,1} \frac{t}{T}\right) + \alpha_g \exp\left(\beta_{g,2} \frac{P_t}{P_{\text{goal}}}\right) \right).$$

In this way, the agent is rewarded more the earlier it reaches the goal and the higher the power with which it reaches the goal. Again, we can use $\alpha_g \in (0,1)$ to weigh the importance of both of these factors, $\beta_{g,1}, \beta_{g,1} \in \mathbb{R}_{>0}$ to determine the reward functions' curvature, and $A_g \in \mathbb{R}_{>0}$ to weigh the goal reward in comparison to the other two.

So, in total, we have

$$r_t = \begin{cases} -A_f \left((1-\alpha_f) \exp\left(-\beta_{f,1} \frac{t}{T}\right) + \alpha_f \exp\left(-\beta_{f,2} \frac{P_t}{P_{\text{fail}}}\right) \right) & \text{if } P_t < P_{\text{fail}} \\ A_g \left((1-\alpha_g) \exp\left(-\beta_{g,1} \frac{t}{T}\right) + \alpha_g \exp\left(\beta_{g,2} \frac{P_t}{P_{\text{goal}}}\right) \right) & \text{if } P_t > P_{\text{goal}} \\ \frac{A_s}{T} \left((1-\alpha_s) \exp\left(\beta_s (P_t - P_{\text{goal}})\right) + \alpha_s \left(P_t - P_{\text{min}}\right) \right) & \text{else} \end{cases}.$$

In order to make sure it is worse to stay just below the goal power than to reach the goal and better to stay just above the fail power than to fail, we choose $A_f, A_g \geq A_s$. In Appendix B.4, we discuss simplified versions of this reward.

5.3 Virtual Testbed

We design a virtual testbed to test different environment hyperparameters and algorithms in a shorter time. By scanning through each of the actuator axes separately and measuring the power, we get a Gaussian distribution on each of the axes, which we fit with a Gaussian. We can then combine these Gaussians by multiplying them and setting an amplitude, and get

$$P(x_{m1}, y_{m1}, x_{m2}, y_{m2}) = A \cdot \exp\Bigg(-\frac{1}{2} \Bigg(\left(\frac{x_{m1} - \mu_{x1}}{\sigma_{x1}} \right)^2 + \left(\frac{y_{m1} - \mu_{y1}}{\sigma_{y1}} \right)^2 + \left(\frac{x_{m2} - \mu_{x2}}{\sigma_{x2}} \right)^2 + \left(\frac{y_{m2} - \mu_{y2}}{\sigma_{y2}} \right)^2 \Bigg) \Bigg)$$

Table 5.3: **Fit parameters:** The power as a function of the position of each actuator was fitted by a Gaussian. Their standard deviations, means, and assumed amplitude can be seen here in the number of actuator steps (incl. fitting error, except for the assumed amplitude).

A	μ_{1x}	σ_{1x}	μ_{2x}	σ_{2x}	μ_{1y}	σ_{1y}	μ_{2y}	σ_{2y}
0.92	5470785	11994	5461786	12769	5573194	19145	5178016	17885
	±41	±34	±46	±37	±40	±33	±47	±39

for the power as a function of the mirror positions with the parameters given in Table 5.3. We chose $A = 0.92$ as that was the highest value observed by us until then. The highest amplitude we observed by now is slightly higher with 0.93 ± 0.02 but still within the measurement uncertainty.

We then use this function to build a virtual testbed in which we model the environment using the above fit. This virtual testbed has one big downside: It does not include the noise present in the experiment, especially in the actions. To get a better noise model, we could, e.g., sample from the noise observed during the dead-zone characterization. We decided not to do this, as more characterization is needed to describe the noise accurately. The simple testbed without noise still helps immensely with environment tuning and picking algorithms. It turns out that an agent trained fully on this virtual testbed still can fiber couple to a certain degree in the experiment, as we will see in Section 5.4.3.

We implement the environment using Gymnasium [338] and use StableBaselines3 [340] for the algorithmic implementation with standard hyperparameters (see Appendix B.1). In all figures, we include error bars or bands of size 2σ, consisting of standard deviations of several runs, which for the return are calculated using ewm.std() from pandas [341, 342], and the (smoothed) mean. If the x-axis says "training steps (rounded to...)," we mean that before plotting and calculating the mean or standard deviation, we had to round the training steps as the saved values are different for each training run.

In this section, we test different environment parameters and algorithms. The usual figure of merit for choosing parameters is the return in dependence of the training step. Some parameters, however, appear in the reward, making it impossible to use the return as a figure of merit. Instead, in those cases, we test the agent every 10^4 training steps for 100 episodes and note the probability of reaching the goal, failing, and the average power at the end of the episodes. Mainly, we use the probability of reaching the goal as the figure of merit but also take the others into account. Only the figures of merit used in the decision process are shown here. Each experiment in the virtual testbed is run 5 times.

In each test, the parameters that are not explicitly given are specified in Tables 5.1 and 5.2. If not mentioned otherwise, we use $P_{\text{goal}} = 0.85$ and TQC.

5.3.1 Reward Hyperparameters

First, we varied the pre-factors of the fail and goal reward $A_f, A_g = 10, 100, 1000$ for different goal powers $P_{\text{goal}} = 0.75, 0.8, 0.85, 0.9$. Figure 5.2 shows the probability of reaching the goal against the training step for 10^5 training steps in total. As one can see, the optimal pre-factors heavily depend on the chosen goal power P_{goal}. For $P_{\text{goal}} = 0.75$, $A_g = 100$ and $A_f = 1000$ perform best. In contrast to that, except for the very start and end, for

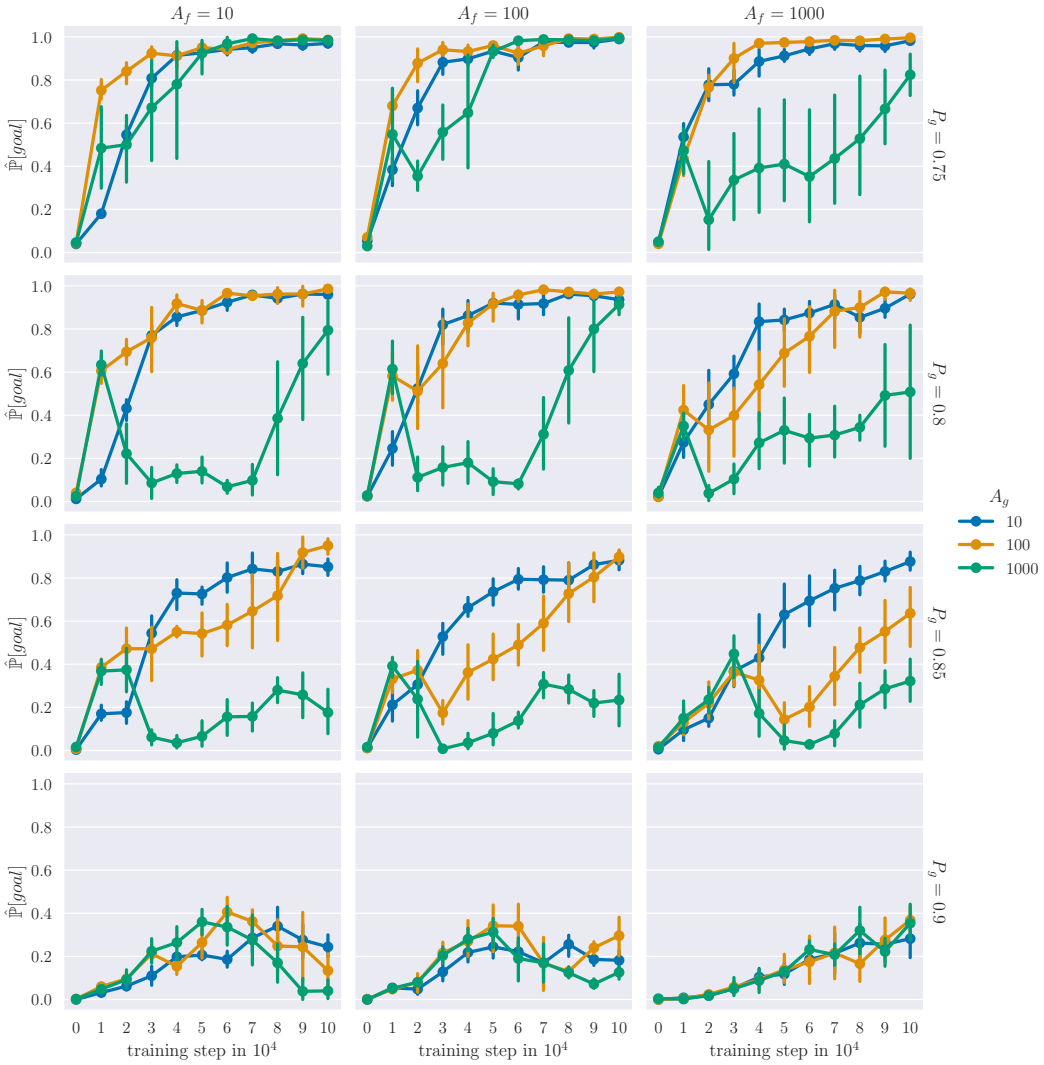


Figure 5.2: **Prefactor tuning results.** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different reward pre-factors A_f, A_g and goal powers P_g (short for P_{goal}) using the parameters in Table 5.1 and TQC. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

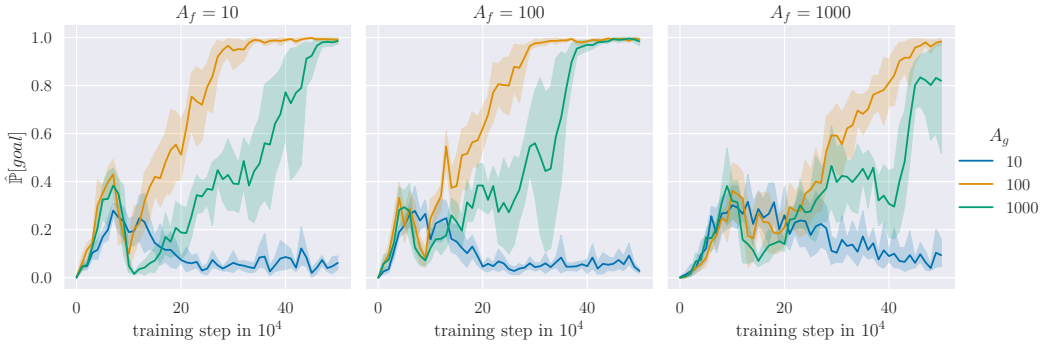


Figure 5.3: **Prefactor tuning for $P_{\text{goal}} = 0.9$.** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different reward pre-factors A_f, A_g using the parameters in Table 5.1, $P_{\text{goal}} = 0.9$ and TQC. The error bands have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

$P_{\text{goal}} = 0.85$, $A_g = 10$ achieves the best results. For $P_{\text{goal}} = 0.9$, the probabilities of reaching the goal stay low, so we also perform the same tests again with a total of $5 \cdot 10^5$ training steps. The results are shown in Figure 5.3.

Generally speaking (except for $P_{\text{goal}} = 0.75$), we can discard $A_f, A_g = 1000$ from our candidate list. For $P_{\text{goal}} = 0.75, 0.8$, $A_g = 100$ outperforms the others. In the first 10^5 training steps for $P_{\text{goal}} = 0.85, 0.9$ this either is not so clear anymore or $A_g = 10$ achieves the best outcome. However, at the end, we can see the $A_g = 100$ curve slowly overtaking the $A_g = 10$ curve, especially for $P_{\text{goal}} = 0.85$. Considering the plot with $P_{\text{goal}} = 0.9$ and $5 \cdot 10^5$ training steps, we again see that $A_g = 100$ performs best. Additionally, $A_f = 100$ yields slightly better results than the other two.

The step reward is especially important in the early stages of training, and the fail and goal rewards become more and more important in the later stages of training (see e.g. [47]). This can also be seen in Figures 5.2 and 5.3: For $P_{\text{goal}} \geq 0.85$, at some of the earlier stages of training, $A_f = 10$ or $A_g = 10$ perform better than $A_f = 100$ and $A_g = 100$, which perform better in later stages. This led us to investigate what happens if we decrease the step reward, i.e., A_s over time. Starting from $A_s = 100, 10$, we decreased A_s linearly over the course of the first $10^5, 2 \cdot 10^5$ training steps to $10, 0$. The results are shown in Figure 5.4 (a) for $P_{\text{goal}} = 0.85$. In both cases where we started with $A_s = 10$ and either held the value constant or decreased it to 0 over the first 10^5 training steps, the probability of reaching the goal rose to approximately 1 but the performance suffers from a small plateau around 10^4 to $7 \cdot 10^4$ training step. On the other hand, leaving the value constant at 100 led to a better performance in the early stages of training, but the probability of reaching the goal did not reach as high values after $2 \cdot 10^5$ training steps. By far, the best results can be seen when we start our training with $A_s = 100$ and decrease its value over the first 10^5 or $2 \cdot 10^5$ training steps to 10 or 0. Between the three tested combinations, there is not much of a difference. For $P_{\text{goal}} = 0.9$, this trick does not work as well anymore, as discussed in Appendix B.2.

Except for β_{g2}, the other reward parameters do not have that much of an effect. Because of that, they are discussed in Appendix B.2. For the other tests, we use the parameters in Table 5.2.

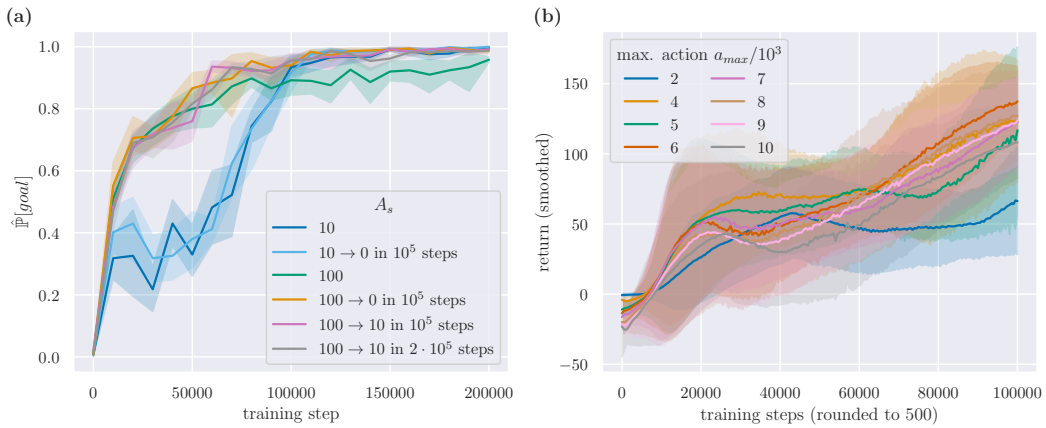


Figure 5.4: **Prefactor changes over time, maximum action.** Panel (a) shows the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different pre-factors of the step reward that change over the course of training. In particular, A_s is decreased linearly from a starting value of $10, 100$ to an ending value of $0, 10$ over the course of the first $10^5, 2 \cdot 10^5$ training steps. Panel (b) shows the return against the training step for different values of the maximum action $a_{\max}$. Both panels use the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. The error bands have a size of 2σ in each direction. For Panel (a), σ is the standard deviation of the five tests. For Panel (b), σ represents a composite standard deviation, combining the standard deviation calculated across five training runs grouped into buckets with the training step rounded to 500, along with the standard deviation of the corresponding smoothed values.

5.3.2 Action

We also tested different maximum actions $a_{\max} = (2, 4, 5, 6, 7, 8, 9, 10) \times 10^3$. Since this parameter does not appear in the reward, we can simply take the return as a figure of merit. The results are shown in Figure 5.4 (b). As one can see, for more than $7 \cdot 10^4$ training steps, the return is highest for $a_{\max} = 6 \cdot 10^3$, which is approximately half of σ_{1x} or σ_{2x}. Before that, $a_{\max} = 4 \cdot 10^3$ performs better. In the experiment, we decided on $a_{\max} = 6 \cdot 10^3$ as the action noise usually leads to the action being smaller, not larger than anticipated.

5.3.3 Episode Length

Now, consider the maximum episode length T. This parameter again appears in the reward, so we employ the same method as for the reward parameters. We evaluated the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different maximum episode lengths $T = 5, 10, 20, 30, 40, 50$. The optimal episode length could depend on the maximum action, as for very small actions, it would not be possible to reach the goal in the same number of steps as for large actions. Hence, we also tested $T = 20, 30$ for two different maximum actions $a_{\max} = 2 \cdot 10^3, 1 \cdot 10^4$. The results are shown in Figure 5.5. Independent of the maximum action $a_{\max}$, we find the higher the episode length, the higher the probability of reaching the goal. This is only logical as the agent has more time to reach the goal. On the other hand, the longer the episodes, the fewer resets are happening in a certain time frame, which leads to the agent seeing fewer different starting conditions. Starting from many different positions in a short time is especially important in the experimental environment,

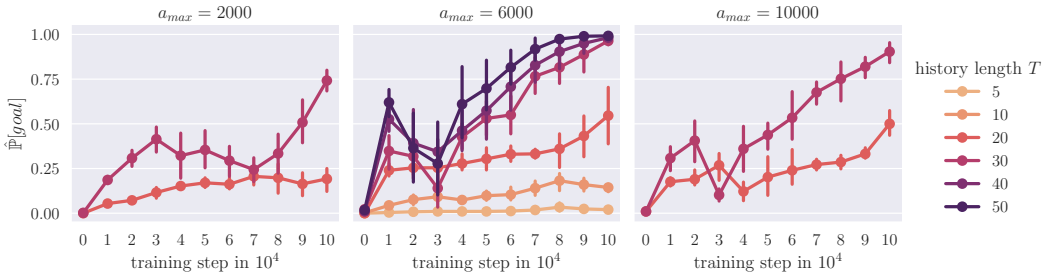


Figure 5.5: **Episode length.** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different maximum episode lengths T and maximum actions $a_{\max}$ using the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. The error bars have a size of 2σ in each direction where σ is the standard deviation of the five tests.

as each step takes approximately a second. So, we want to balance the two. Between $T = 20$ and $T = 30$, there is still a big difference in performance, which gets smaller between $T = 30$ and $T = 40$ or even $T = 50$. That is why we decided to set $T = 30$ by default.

5.3.4 Observation

Let us study the observation now. As discussed in Section 5.2.2, we use

$$O_t = \left((P_{k-1}, A_{k-1}, P_{\text{ave},k}, P_{\max,k}, x_{\max,k})_{k=t-n,\dots,t}, P_t \right)$$

as our observation. We want to answer three questions: First, is it useful to include the power P_t, the action a, the average power P_{ave}, maximum power $P_{\max}$, and its relative position $x_{\max}$ in the observation? Second, what is the optimal history length n? Third, would it be useful to include the absolute position in the observation?

Regarding the first question, we evaluate the return against the training step for different configurations of the observation, i.e., using the full observation and leaving out P_t, P_{ave}, $P_{\max}$, $x_{\max}$, or a. The results are shown in Figure 5.6 (a). As we can see, the most important part of the observation is the performed action a. The second most important part is the relative position of the maximum $x_{\max}$. Each individual power in the observation, i.e., P_{ave}, $P_{\max}$, and P_t, is not that important, as the returns when leaving out one of them is very close to the one with the full observation. While leaving out P_{ave} and P_t has a small, overall negative effect, leaving out $P_{\max}$ actually helps during the early stages of training and only has a slightly negative impact in the later stages.

For the second question, we consider the return against the training step for different history lengths $n = 1, \dots, 6, 30$. The results are shown in Figure 5.6 (b). We can easily see that $n = 3, 4$ perform better in the end than the others (although $n = 2, 5, 6$ are close). However, $n = 4$ performs better around $2 \cdot 10^4 - 5 \cdot 10^4$ training steps, which is why we choose $n = 4$. It makes sense that observation lengths of approximately four perform best, as we have four actuators and one scalar sensor. After four actions, if we assume that the action vectors are linearly independent, one could be able to find out analytically which actuator had which effect. Hence, it is reasonable that the RL agent can also determine the next best action better than with $n = 2, 1$. On the other hand, if $n > 4$, the agent does not get much more information that would be analytically helpful, but the observation would get higher

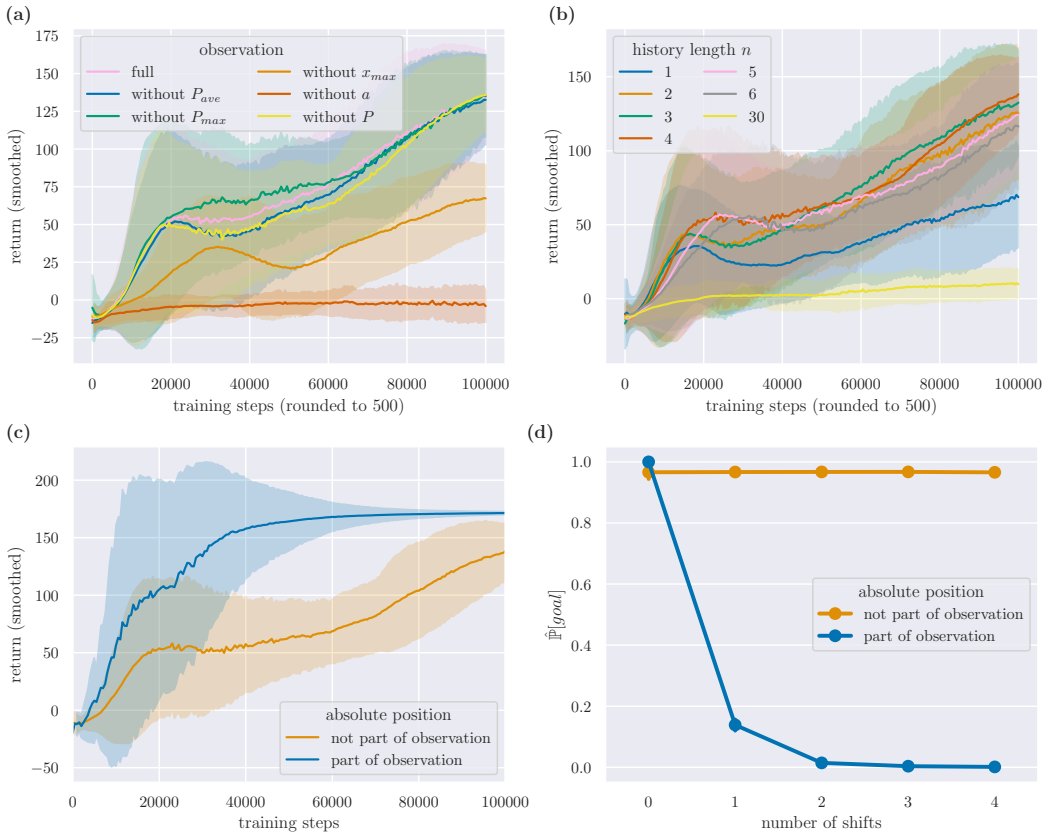


Figure 5.6: **Observations.** Panel (a)-(c) show the return against the training step for different observations using the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. Panel (a) shows what happens if we leave out different parts of the observation with $n = 4$. In Panel (b), we vary the history length $n = 1, \ldots, 6$. Panel (c) shows the return if we include the absolute actuator positions in the observation or not. These ten agents (five with or without absolute positions in the observation each) are then tested in environments where 1σ-shifts are applied to the optimal positions, i.e., $\mu_i' = \mu_i \pm \sigma_i$, of k actuators ($k = 0, \ldots, 4$). Each of the combinations is tested 100 times. Panel (d) shows the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against k. The error bars and bands have a size of 2σ in each direction. For Panels (a)-(c), σ represents a composite standard deviation, combining the standard deviation calculated across five training runs grouped into buckets with the training step rounded to 500, along with the standard deviation of the corresponding smoothed values. For Panel (d), σ is the standard deviation of the tests.

dimensional. Then, the underlying NNs of the agent would have more parameters and hence require more samples of the agent-environment interaction. Therefore, it is plausible that $n = 4$ performs best numerically.

To answer the third question, we train five agents, each in environments without and with the absolute position in the observation. The return against the training step is shown in Figure 5.6 (c). Each of these agents is then tested in environments in which the optimal positions μ_i are shifted by $\pm\sigma_i$, i.e., $\mu_i' = \mu_i \pm \sigma_i$, for k actuators where $k \in \{0, \dots, 4\}$. Each of the possible combinations is tested 100 times. Figure 5.6 (d) shows the probability of reaching the goal against the number of shifts k. During training, the return is much higher if we include the absolute position in the observation. The probability of reaching the goal is also slightly higher with them if the optimal position stays the same. However, if the optimal position of only one actuator is shifted, the probability of reaching the goal plummets. On the other hand, if we do not use the absolute position as part of the observation, the probability of reaching the goal stays the same. An agent not using the absolute position as part of the observation can, therefore, generalizes well to different optimal positions. As the fiber coupling agent is needed in the experiment in case the experiment before the motorized mirrors is aligned differently, it has to be able to reach the goal regardless of changed optimal positions. Hence, we do not use the absolute actuator positions as part of the observation and can thus realign the experiment even if another part was aligned differently.

5.3.5 Goal Power

We can choose the goal power depending on our needs. We now investigate at which point the training converges for different goal powers. Figure 5.7 (a) shows the return against the training steps for different goal powers $P_{\text{goal}} = 0.8, 0.85, 0.86, ..., 0.91$. As expected, the higher the goal power, the lower the value to which the return converges and the later the point of convergence. Furthermore, for the same gap between two different goal powers, the gap between the two returns associated with these goal powers gets bigger with higher goal powers.

This makes it much harder to train on high goal powers like $P_{\text{goal}} = 0.9$, especially in the experiment where each environment step takes approximately a second. We now want to find out if it could make sense to pre-train the agent on lower goal powers and raise it over the course of training. Starting from a goal power of $P_{\text{start, goal}} = 0.5, 0.7, 0.8, 0.85, 0.875, 0.9$, we raised the goal power to $P_{\text{end, goal}} = 0.9$ over the course of 10^5 training steps either linearly, i.e., increasing the goal power by a small amount in each training step, or in a step-wise manner, i.e. increasing it every 10^4 training steps by a bigger amount. Again, this is a parameter appearing in the reward, so we cannot take the return as our figure of merit. Figure 5.7 (b) shows the probability of reaching the goal $P_{\text{end, goal}} = 0.9$ after 10^5 training steps against the starting goal power $P_{\text{start, goal}}$ for both ways of increasing the power. We can see that it can be beneficial to increase the goal power in a step-wise manner for $P_{\text{start, goal}} \in [0.7, 0.85]$, especially $P_{\text{start, goal}} = 0.85$. This can be seen as an instance of *curriculum learning*, where an agent is trained on successively more difficult tasks [343].

5.3.6 Reset Methods

A way often used for resetting is to choose parameters at random in a given interval. We now want to compare this method to the reset method discussed in Section 5.2.3 for different values of l, the number of resets after which we move to the neutral positions. First, we

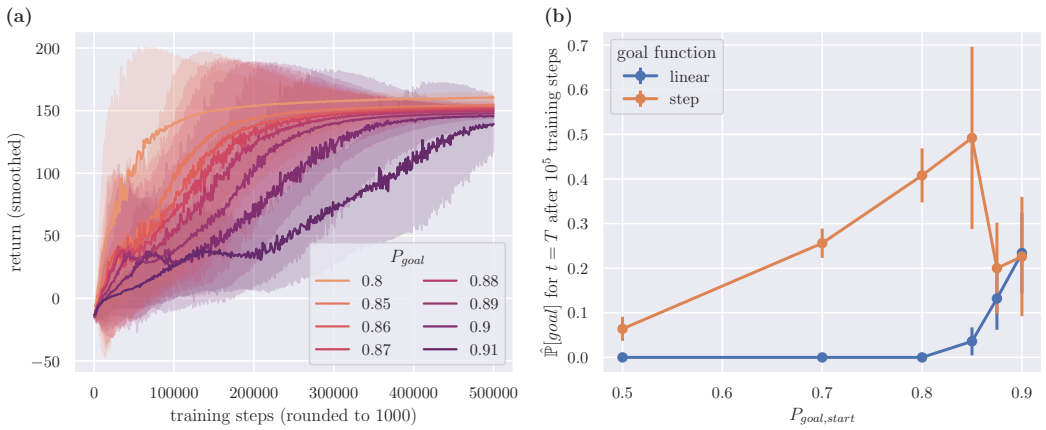


Figure 5.7: **Goal Power.** Panel (a) shows the return against the training step for different values of the goal power P_{goal}. Panel (b) shows the probability $\hat{\mathbb{P}}[goal]$ of reaching the goal $P_{\text{goal}} = 0.9$ after 10^5 training steps against the starting goal power $P_{\text{goal, start}}$. Thereby, the goal power of the environment is increased either linearly (blue) or in a step-wise manner (orange) from $P_{\text{goal, start}}$ to $P_{\text{goal}} = 0.9$ over the course of the 10^5 training steps. Both panels use the parameters in Tables 5.1 and 5.2 and TQC. The error bars and bands have a size of 2σ in each direction. For Panel (a), σ represents a composite standard deviation, combining the standard deviation calculated across five training runs grouped into buckets with the training step rounded to 1000, along with the standard deviation of the corresponding smoothed values. For Panel (d), σ is the standard deviation of the tests.

compare the returns for the different reset methods. This is shown in Figure 5.8 (a). For the interval reset method, we choose a radius of $2.1 \cdot 10^4$ around the mean to get a similar median of starting power (in comparison with our reset method). For $l = 1$ and the interval method, consecutive episodes are independent of each other – at least in the virtual testbed. This is not the case anymore for $l > 1$ or in the experiment. The mean of the return is quite similar for the different parameters of l, and we cannot make out a trend of changes in the return with rising l. The return when using the interval method stays below the others for the radius we have chosen and the form of the curve is slightly different. However, it is unclear if that is due to the radius chosen.

For better comparison, we also denote the actuator positions and starting powers after resetting when testing the agent for 100 consecutive episodes. The power distribution and actuator position distribution for some of the reset methods are shown in Figure 5.8 (b) and (c), respectively. In Panel (b), we can see that the starting power for the interval method is distributed over nearly the full range $[0, 0.9]$ and is centered around $[0.15, 0.4]$. In contrast, the other reset methods yield starting powers in the range $[0.19, 0.82]$ and are centered around $[0.19, 0.35]$. Their distribution is more skewed to 0.2, with medians still being slightly smaller than for the interval method. Still, the return stayed smaller for the interval method, which could be due to the agent directly failing in some cases. The median and 75^{th} percentile for $l = 1$ are slightly bigger than for $l > 1$ at the start of training, although – at least for $l = 10$ – this difference gets smaller during training.

In Panel (c), we can see the distribution over starting actuator positions of the first mirror for the interval method, $l = 1$ and $l = 10^5$. The star marks the mean of the Gaussian distri-

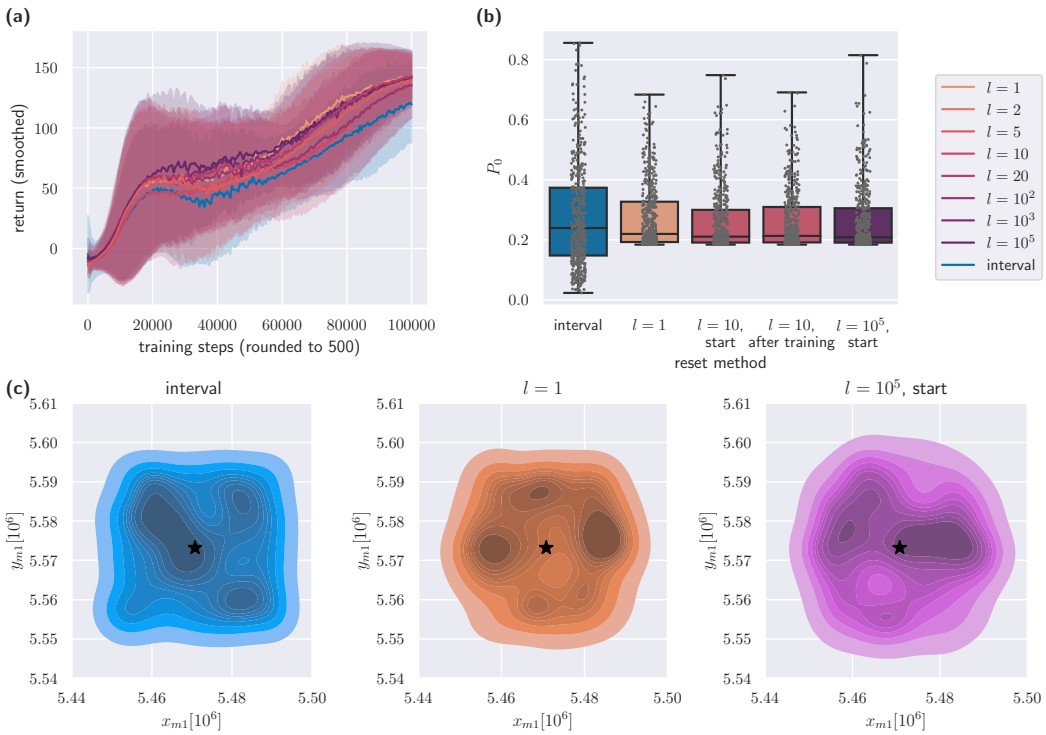


Figure 5.8: **Reset methods.** Panel (a) shows the return against the training step for different reset methods. Panel (b) shows the starting power of 100 consecutive episodes against the used reset method. For $l > 1$, this depends on the agent used, which is why we also state if we use the agent at the start of training or after training in that case. Panel (c) shows the starting actuator positions of mirror 1 for the 100 episodes and different reset methods. The stars show the mean of the Gaussians as described in Table 5.3. For all plots we use the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. The error bands in Panel (a) have a size of 2σ in each direction, where σ represents a composite standard deviation, combining the standard deviation calculated across five training runs grouped into buckets with the training step rounded to 500, along with the standard deviation of the corresponding smoothed values.

butions used for the virtual testbed. As expected, for the interval method, the distribution is more quadratic with the highest probability in the middle. For our reset method, the distributions become more toroidal. For $l = 1$, we can see that the highest probabilities can be found on a ring around the mean of the Gaussians. This is to be expected as we designed our reset method in a way that our episodes do not often start with powers above the goal or below the failing power. For $l = 10^5$ at the start of training, we see that this distribution gets a little wider, and there are fewer and bigger areas of the highest probability. This is probably due to the episodes not being independent of each other. However, this does not have a noticeable impact on the return. We nevertheless have to keep in mind that not returning to the neutral positions can lead to some areas not being visited as often as others.

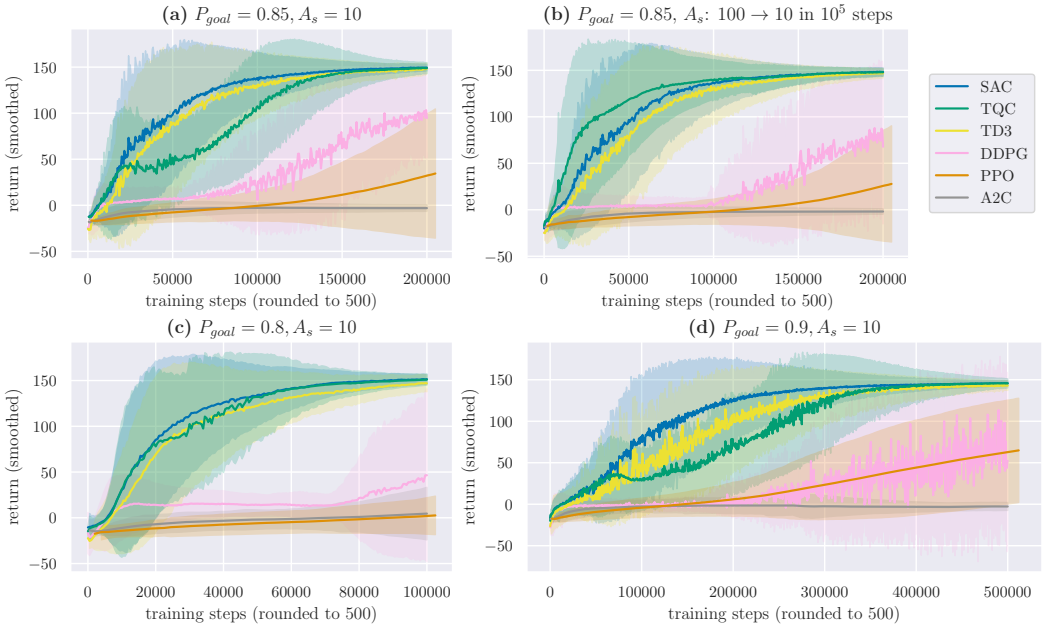


Figure 5.9: **Algorithms.** All plots show the return against the training step for different algorithms using the parameters in Tables 5.1 and 5.2. For the different panels, we use different goal powers, and for Panel (b), we decrease the pre-factor of the step reward A_s from 100 to 10 in the first 10^5 steps. The error bands have a size of 2σ in each direction, where σ represents a composite standard deviation, combining the standard deviation calculated across five training runs grouped into buckets with the training step rounded to 500, along with the standard deviation of the corresponding smoothed values.

5.3.7 Algorithms

Last, but not least, we tested out different RL algorithms, in particular SAC, TQC, TD3, DDPG, PPO and A2C with standard hyperparameters as explained in Appendix B.2, in our environment for $P_{\text{goal}} = 0.8, 0.85, 0.9$. As it was beneficial to decrease the step reward over the course of training when using TQC, we also tested how or if this would change the comparison between algorithms for $P_{\text{goal}} = 0.85$. The results are shown in Figure 5.9.

At the start of training, all plots show the worst performance for A2C and PPO. This is to be expected, as both these algorithms, in contrast to the others, do not use replay buffers, which makes these algorithms less sample-efficient. They are closely followed by DDPG. For higher goal powers, PPO catches up or starts catching up with DDPG in the later stages of training. SAC, TQC, and TD3 show a much better performance. SAC performs consistently slightly better than TD3. For a constant $A_s = 10$, TQC starts off and ends with about the same performance as SAC but suffers from a drop in performance in the middle stages of training. The higher the goal power, the more significant this drop is. As we discussed in Section 5.3.1, starting from $A_s = 100$ and decreasing it during training helps significantly in dealing with this drop when using TQC. We can also see this in Figure 5.9 (b). Interestingly, this has the most effect on TQC, which leads to TQC outperforming SAC, and PPO and DDPG to perform worse.

Overall, SAC achieves the best results for this task when considering a constant $A_s = 10$ and TQC when considering a start value of $A_s = 100$ and decreasing it linearly to $A_s = 10$ over the course of the first 10^5 training steps, at least for $P_{\text{goal}} = 0.85$.

5.4 Experimental Results

We used the results from the virtual testbed to select algorithms and shape the environment by choosing the parameters in Tables 5.1 and 5.2 for the experimental environment. Now, we present the results from the experimental environment appearing in [105]. To run the experiments, we use, in addition to the usual packages [344–347], PyLabLib, Thorlabs Kinesis, PyVisa, Keysight Connection Expert and safe-exit [348–352] for communication with the experiment. All training and testing runs presented here took around 20 days in total (with an NVIDIA GeForce RTX 4070 GPU). The training runs for $P_{\text{goal}} \leq 0.87$ took around 20 hours or $4 \cdot 10^4$ training steps. For $P_{\text{goal}} = 0.9$, this went up to nearly 4 days or $2 \cdot 10^5$ training steps. All training runs were performed with the parameters in Tables 5.1 and 5.2.

5.4.1 Algorithms

For $P_{\text{goal}} = 0.85$, we tested both SAC and TQC two times each. The results are shown in Figure 5.10 (a). Although the two are very similar, TQC is performing slightly better in the experiment for a goal power of $P_{\text{goal}} = 0.85$. Therefore, we carried out the following experiments presented here using TQC. However, we only performed this comparison for $P_{\text{goal}} = 0.85$, and not $P_{\text{goal}} = 0.9$. As we know from Section 5.3.7, the difference in performance for TQC and SAC gets more favorable for SAC with rising goal power. Hence, a promising future direction is to test the performance of SAC and TQC for a higher goal power like $P_{\text{goal}} = 0.9$.

5.4.2 Goal Powers

We also performed experiments using TQC and goal powers $P_{\text{goal}} = 0.85, 0.86, 0.87, 0.88, 0.9$ until the training runs started to converge. The return against the training step is shown in Figure 5.10 (b). For $P_{\text{goal}} \leq 0.87$, the training converges around $4 \cdot 10^4$ training steps, which corresponds to approximately 20 hours. For $P_{\text{goal}} = 0.88$, this takes significantly longer, i.e. nearly $6 \cdot 10^4$ training steps or 30 hours. For $P_{\text{goal}} = 0.9$, we even see a pronounced dip in performance, which we also saw in the virtual environment with TQC. This training takes about $2 \cdot 10^5$ training steps or nearly 4 days to stabilize. Additionally, as expected, the return does not reach as high values anymore.

Compare the plots for the virtual testbed in Figure 5.7 (a) and the experiment in Figure 5.10 (b). The trend in both plots is similar, but in the virtual testbed, the training generally reaches higher values for the higher goal powers and stabilizes later. Furthermore, in the virtual testbed, the return already has a small drop for $P_{\text{goal}} = 0.85$ and not only starting from $P_{\text{goal}} = 0.88$ as in the experiment. The fact that the return for higher goal powers $P_{\text{goal}} \geq 0.88$ already stabilizes at much lower values could be due to the action noise, making it harder for the agent to reach high goal powers. A possible explanation for the later convergence and larger drop in performance in the virtual testbed for the same goal power is that we chose the maximum possible power A in the virtual testbed to be too low. When designing the virtual testbed, we had to choose the amplitude A of the Gaussian, which serves as the base for our virtual testbed. As fibers are not loss-free, this amplitude

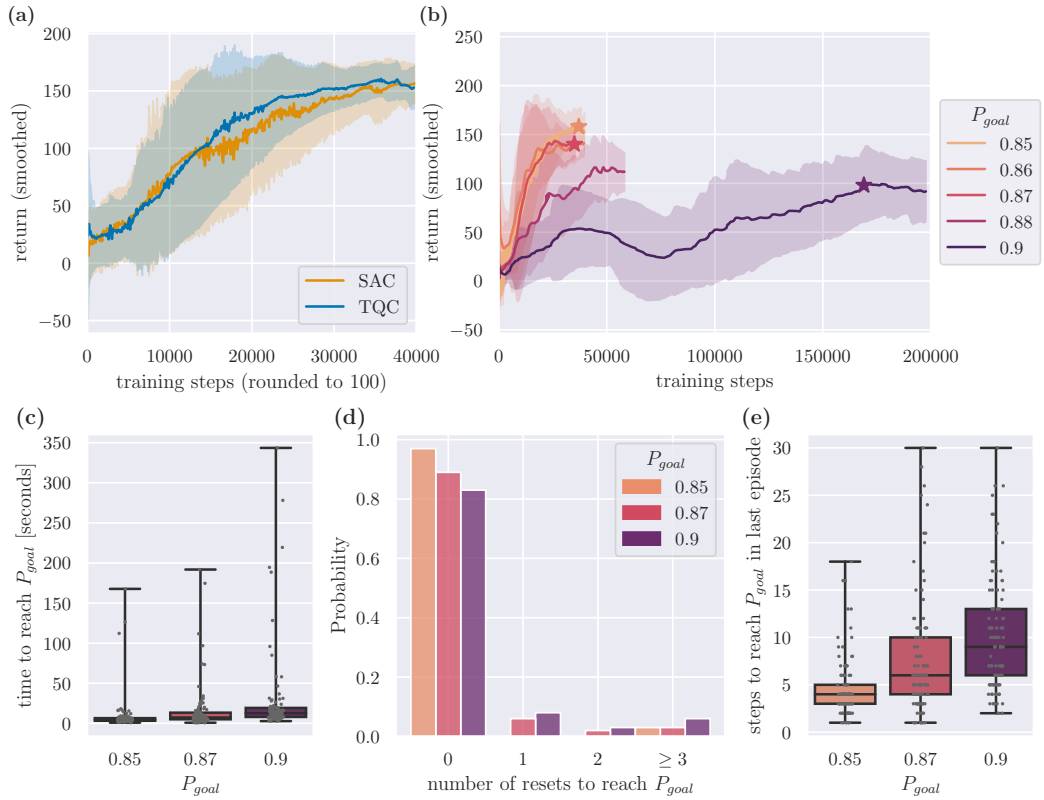


Figure 5.10: **Different algorithms and goal powers in the lab.** Panels (a) and (b) show the return against the training steps for training in the experiment with the parameters in Tables 5.1 and 5.2. In Panel (a), we use $P_{\text{goal}} = 0.85$ and SAC or TQC. For both algorithms, we performed two training runs. In Panel (b), we use TQC and vary the goal power P_{goal}. We tested the agents marked with stars 100 times in the experiment by resetting the environment, starting a timer, and letting the agent try to reach the goal. If the agent did not reach the goal in a certain episode, we reset and keep the timer running. This is repeated until the agent reaches its goal. Panel (c) shows the time each agent needed to reach their goal, Panel (d) shows the number of resets each agent needed to do so, and Panel (e) shows the number of steps it took each agent to reach the goal in the last episode. The error bands in Panels (a) and (b) have a size of 2σ in each direction. In Panel (a), σ represents a composite standard deviation, combining the standard deviation calculated across two training runs grouped into buckets with the training step rounded to 100, along with the standard deviation of the corresponding smoothed values. In Panel (b), σ is given by the standard deviation arising from smoothing. The whiskers of the boxplots in Panels (c) and (e) show the 0^{th} and 100^{th} percentile.

had to be smaller than 1. On the other hand, it had to be at least as high as the maximum observed power. Conservatively, we chose it as the maximal power observed until the design of the testbed, which led to $A = 0.92$. However, by now, we also observed values of 0.93 ± 0.02. This is a reasonable explanation for the difference in the figures. Nonetheless, the virtual testbed is helpful in the selection of environment parameters and algorithms.

Now, consider the testing results. We tested the agents marked with stars in Figure 5.10 (b) 100 times each by resetting the environment, starting a timer, letting the agent try from there, and resetting the agent in between while keeping the timer running if it did not reach the goal in a given episode. The results are displayed in Figure 5.10 (c)-(e). Panel (c) shows the time it took each agent to reach its goal. As expected, this time goes up with the goal power. Still, each agent reliably reaches its goal. Panel (d) shows the probability of each agent taking a certain number of resets to reach the goal. We can see that the probability of reaching the goal in the first episode decreases with an increasing goal power while the probability of needing more episodes increases. However, even for $P_{\text{goal}} = 0.9$, the probability of reaching the goal in the first episode is more than 80%. Panel (e) shows the number of steps it took each agent to reach the goal in the last episode. Again, as expected, this number rises with the goal power. Hence, both the increase in time for reaching the goal (Panel (c)) and the decrease in the return (Panel (b)) with rising goal power are due to both an increasing number of resets needed (Panel (d)) and the number of an increasing number of time steps needed in each episode (Panel (e)).

5.4.3 Pre-training

As the standard agent needs many training steps for high goal powers like $P_{\text{goal}} = 0.9$, we want to see if pre-training helps us. Pre-training means that we train an agent first in a different environment and then transfer it to the environment we are interested in to train the agent further. When transferring the agent, we can either keep the replay buffer of the old environment or not. The environments we pre-trained our agent on were either the lab environments with lower goal power or the virtual testbed.

Pre-training on the virtual testbed We trained a TQC agent in the virtual testbed with $P_{\text{goal}} = 0.9$ for $5 \cdot 10^5$ training steps. Then, we further trained the agent in the lab environment for $2.18 \cdot 10^5$ training steps.

Pre-training on lower goal powers We trained two TQC agents in the lab first with $P_{\text{goal}} = 0.85$ for $3.8 \cdot 10^4$ training steps, then with $P_{\text{goal}} = 0.875$ for $2.5 \cdot 10^4$ training steps, then with $P_{\text{goal}} = 0.89$ for $3.5 \cdot 10^4$ training steps, and finally with $P_{\text{goal}} = 0.9$ for $1.14 \cdot 10^5 - 1.26 \cdot 10^5$ training steps. In one of them, we kept the replay buffer at each change in goal power except the last. In the other one, we reset the replay buffer at each change in goal power.

Figure 5.11 shows the training and testing results obtained in the same way as discussed in Section 5.4.2.

Panel (a) shows the return against the training step for these three agents and the one without pre-training for comparison. Both the agent pre-trained on lower goals with deleted replay buffer and the agent pre-trained on the virtual testbed performed better than the one without pre-training. They perform similarly to each other. However, for the one pre-trained on the virtual testbed, the person using this algorithm would need to spend time

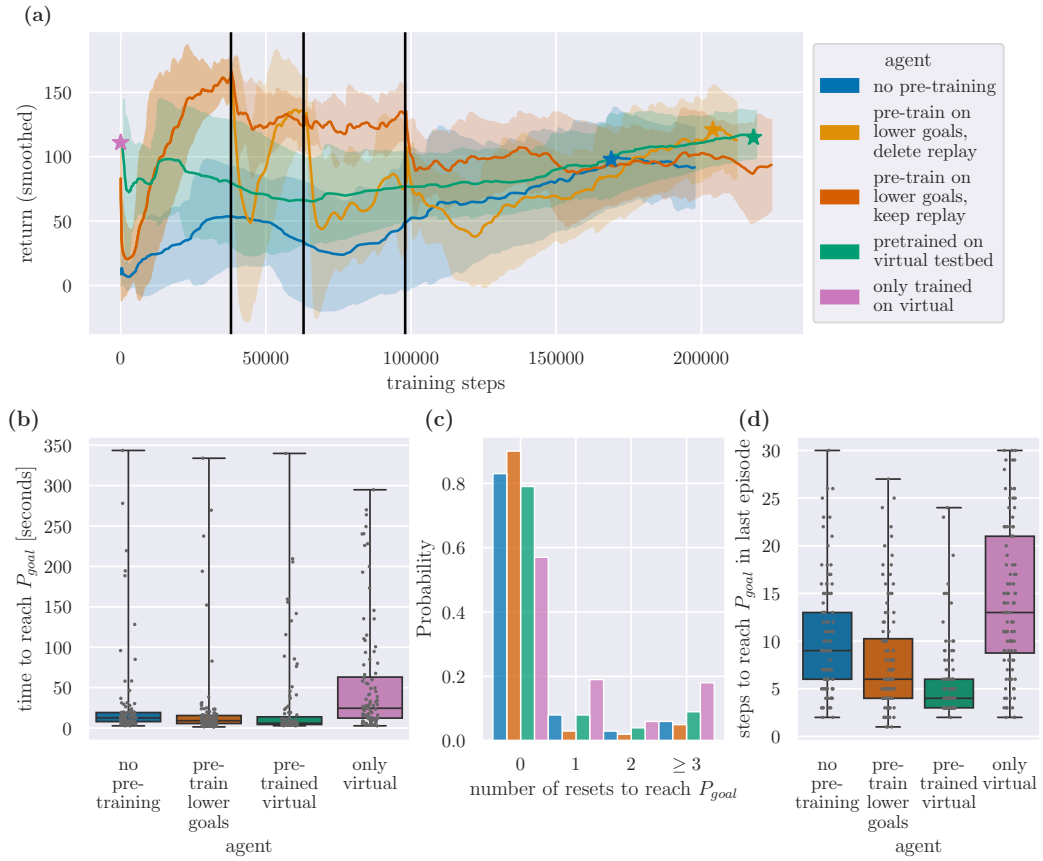


Figure 5.11: **Different pre-training regimes in the lab.** Panel (a) shows the return against the training steps for training in the experiment with the parameters in Tables 5.1 and 5.2 and TQC. Except for one agent (dark blue), which was fully trained with $P_{\text{goal}} = 0.9$ in the lab, each agent is pre-trained in a certain manner. The green agent was first pre-trained in the virtual testbed and only then further trained in the lab, in both cases with $P_{\text{goal}} = 0.9$. Both orange ones were pre-trained on lower goal powers, first with $P_{\text{goal}} = 0.85$ for $3.8 \cdot 10^4$ training steps, then with $P_{\text{goal}} = 0.875$ for $2.5 \cdot 10^4$ training steps, then with $P_{\text{goal}} = 0.89$ for $3.5 \cdot 10^4$ training steps, and finally with $P_{\text{goal}} = 0.9$ for $1.14 \cdot 10^5 - 1.26 \cdot 10^5$ training steps. These changes in goal power are marked with black vertical lines. For the lighter orange, the replay buffer was deleted after each stage except the last one. For the dark orange one, the replay buffer was kept throughout the training. We then tested the agents marked with stars 100 times in the experiment by resetting the environment, starting a timer, and letting the agent try to reach the goal. If the agent did not reach the goal in a certain episode, we reset and keep the timer running. This is repeated until the agent reaches its goal. Panel (b) shows the time each agent needed to reach their goal, Panel (c) shows the number of resets each agent needed to do so, and Panel (d) shows the number of steps it took each agent to reach the goal in the last episode. The error bands in Panel (a) have a size of 2σ in each direction, where σ is given by the standard deviation arising from smoothing. The whiskers of the boxplots in Panels (b) and (d) show the 0^{th} and 100^{th} percentile.

building a virtual testbed, which is not required for pre-training on lower goals in the lab. So, examining the return, we would not recommend building a virtual testbed with the only purpose of pre-training. Note that we would get better results if we included noise in the virtual environment, e.g., by sampling from the dead-zone characterization in Figure 5.1 (c). However, we would then need to spend more time characterizing the noise than it would save us in training time in the laboratory, which is why we decided against pre-training on a virtual testbed with noise.

Panels (b)-(d) again show the testing results using the same metrics as in Figure 5.10 but with different agents. In Panel (b), we can see that all agents trained in the lab reach the goal in a similar time frame. The one without pre-training took the longest time, then the one pre-trained on lower goals, and the one pre-trained on the virtual testbed was the fastest, but these differences are very small. The agent only trained on the virtual testbed takes more time (almost double on average) to couple the light into the fiber but still does so consistently and, on average, similarly fast as the human experimenter [105]. As the training times on the virtual testbed are much shorter, however, it can still make sense to train exclusively on the virtual testbed. Let us now look at Panels (c) and (d) to understand where these differences come from. The agent only trained on the virtual testbed takes the most resets and the most steps in the last episode to reach the goal, which makes it clear that it is the slowest. If we compare the other three, however, we see that regarding the probability of reaching the goal in the first episode, the agent pre-trained on lower goals performs best, then the agent not pre-trained, and the agent pre-trained on the virtual testbed comes in last. Regarding the number of steps to reach the goal in the last episode, the agent pre-trained on the virtual testbed performed best, then the agent pre-trained on lower goals, and then the agent without pre-training. This is interesting because, apparently, the agent pre-trained on the virtual testbed can make up for not reaching the goal in each episode reliably by being very fast if it does so. On the other hand, the agent without pre-training cannot make up for not reaching the goal as fast in each episode by reaching it more reliably in the first episodes.

Overall, from this data, we conclude that pre-training on lower goals (when refreshing the replay buffer after bigger goal jumps) or in the virtual testbed can be very beneficial. However, the latter only makes sense if such a virtual testbed already exists and does not have to be designed for only this purpose because it does not drastically reduce training time. If we did not have enough time to train on the experiment, training only on the virtual testbed can be a possibility. This agent still couples in reliably although it is not as fast as its counterparts trained in the lab.

5.5 Summary and Outlook

We designed an RL environment for experimental fiber coupling, where the agent has to reach a goal of a certain coupling efficiency P_{goal} starting from a small coupling efficiency. Fitting a scan of each actuator axis by a Gaussian, we obtained a virtual testbed. Both the lab environment and virtual testbed were implemented using Gymnasium [338], and as agents, we used the StableBeselines3 [340] implementations with standard hyperparameters as discussed in Appendix B.1. Using the virtual testbed, we carefully tuned the environment and selected RL algorithms. Our main focus was using as few samples as possible, as each environment step takes about 1 second in the lab. We then trained RL agents in the lab environment, using different algorithms, in particular, SAC and TQC, goal efficiencies P_{goal}

or pre-training regimes.

In the lab, we successfully trained TQC agents with different pre-training regimes to reliably reach coupling efficiencies of more than $90\% \pm 2\%$ over the course of approximately 4 days. For reference, the highest efficiency reached by the human experimenter was $92\% \pm 2\%$, and the highest observed during the training of the agents was 93%. Both pre-training on lower goals (when refreshing the replay buffer after bigger goal jumps) and in the virtual testbed were very beneficial for us. For the purpose of spending as little time as possible in the lab, we only recommend the latter if the virtual testbed already exists or is used for other reasons than speeding up the training. Although being much slower, even an agent only trained in the virtual testbed reliably reaches this goal without training in the experiment. Lower goals, i.e., $P_{\text{goal}} \leq 87\%$, were already reached with a training time in the laboratory of less than a day. Although SAC performed better in the virtual testbed, for $P_{\text{goal}} = 85\%$, TQC performed slightly better in the lab.

The two papers using RL experimentally in optical table-top experiments use significantly simpler environments. For the combination of coherent beams in [330], only one actuator is used. The training time for their experiment is much shorter (4 hours). However, the authors show in simulation that with the same number of actuators we use, the training times would quickly rise to $1-2$ days, which is comparable to our training times for moderately high goal efficiencies like $P_{\text{goal}} = 85-88\%$. In [331], the authors use the actuator positions as both their actions and state, which makes the problem an MDP, that is, in principle, fully observable, with an optimal action. Additionally, their action space is only three-dimensional. They claim that their TD3 agent learns in 20 minutes.

The here presented agents learn to deal with the noise in our lab environment. The actuators' imprecision makes it infeasible to use classical methods like gradient descent directly. To use them, we would, for example, need additional feedback loops observing the actual distance traveled by the actuators. With RL, we can avoid them, which simplifies the experiment. The automation of experiments not only frees up time for human experimenters but also makes remote control much easier, which can be crucial in environments like vacuum tanks, cleanroom facilities, underground, or space.

Regarding fiber coupling, future directions could include studying model-based or hybrid algorithms for fiber coupling. This has the upside of probably needing fewer samples in the given environment and the downside of algorithmic implementations not being as readily available as the model-free variants we tried here. Another direction would be testing how well a decay of the dense reward over training steps would work in the lab environment. We could evaluate how well the trained agents do on non-Gaussian modes of light or directly train agents on a variety of modes. Additionally, we could introduce more degrees of freedom by also motorizing the lenses. Furthermore, here we only discuss starting with a small coupling efficiency because, if we had no light, the agent could only do random movements as it cannot get any feedback. It would be interesting if we could start with no light behind the fiber, either by using additional sensors or cameras or by combining RL with algorithms like a grid search. Other types of observations that are not the history, like PID-inspired RL [353], could be explored. We can scale the presented task of fiber coupling to more complex alignment tasks by dividing systems into mirror-mirror-sensor blocks and solving them in sequence. Hence, we have shown everything needed for the spatial alignment of light.

We also see our experiments as a first step for using RL for experimental alignment and control of other optical experiments. An example of a future experiment could be stabilizing a cavity. This is usually done using the Pound-Drever-Hall procedure (see [354] for an introduction) which requires a number of additional components like phase modulators [355], homodyne detectors [356,357], or split detectors [358,359]. Instead, an RL agent could rely solely on the reflection or transmission power, which not only uses fewer components but could also open the door to novel control strategies, e.g., for the phase control of squeezed vacuum states, particularly relevant for on-chip squeezing experiments for quantum information. Current strategies include auxiliary laser fields [360] or introduce unwanted noise [361]. RL has the potential to perform these tasks without additional laser fields or noise.

Data availability The python code and experimental data is available at `https://github.com/ViktoriaSchmiesing/RL_Fiber_Coupling`.

6

Conclusion and Outlook

In this work, we studied the application of machine learning (ML) to quantum mechanical and optical systems. To lay the groundwork for the discussion of dissipative quantum recurrent neural networks (DQRNNs) and the application of reinforcement learning (RL) to fiber coupling, we first introduced key concepts from both classical machine learning (ML) and quantum information (QI) theory.

We began by providing an overview of classical ML, focusing on supervised learning (SL) and RL. After presenting the SL framework for independent and identically distributed (i.i.d.) data, we discussed feed-forward (ff) and recurrent neural networks (NNs), along with their respective training algorithms, as a foundation for understanding DQRNNs. We then introduced the RL framework and detailed the algorithms used later in experimental fiber coupling.

Moving on, we covered essential topics in QI, including quantum states, measurements, composite systems, and channels. We introduced qubits and quantum circuits as quantum analogs to bits and logical circuits, respectively. Furthermore, we discussed relevant norms and distance measures, such as the fidelity, that are crucial for defining cost functions for learning quantum states. The sampling of quantum states and unitaries is critical for applying ML to quantum data, so we discussed the Haar measure. In light of its relevance to DQRNNs, we explored quantum channels with memory. Additionally, we introduced tensor networks (TNs), which are instrumental in the development of some of the classical DQRNN training algorithms. We concluded by exploring quantum machine learning (QML), categorizing it into four types – CC ML, CQ ML, QC ML, and QQ ML – based on the nature of the data (first letter) and the algorithms (second letter) involved. Our focus was on dissipative quantum neural networks (DQNNs) for learning quantum data and its training data, costs, and training algorithms. Furthermore, we examined the specific challenges faced in QML, particularly the issue of barren plateaus, where the optimization landscape flattens exponentially as system size increases, complicating the training process.

We developed a fully quantum recurrent neural network (QRNN) architecture, called dissipative quantum recurrent neural network (DQRNN), which combines the concepts of RNNs and quantum channels with memory. In each run of the DQRNN, we iterate over the underlying DQNN, using a part of the DQNN output in one iteration step as part of the next iteration steps DQNN input. The DQRNN can be viewed as both a recurrent version of

the DQNN and as a quantum channel with memory, where a DQNN replaces the general channel. As a result, together with an initialization, DQRNNs can approximate any causal quantum automaton on qudits, making them universal in that sense.

In a DQRNN run, the full input and output are referred to as the global input and output, while the inputs and outputs at each DQNN iteration are called the local input and output. Unlike classical RNNs, where the global input and output are simply lists of the local inputs and outputs, DQRNNs can represent any quantum channel with memory, resulting in local DQRNN outputs that can be entangled with each other, even if the local inputs are not. Hence, we consider two different kinds of training data: product data, where the global input and target output are product states, and MPS data, where the global input and target output are entangled and can be handled as an MPS.

As our cost function, we use the infidelity for pure target outputs and the squared Hilbert-Schmidt distance for mixed target outputs. We can evaluate the DQRNN's outputs and target outputs either globally or locally by considering local and global costs. The local cost is particularly useful for product data. For both local and global costs and pure target outputs, we present quantum training algorithms. The number of required qudits increases with the width but not the depth of the underlying DQNN or, in the case of using the local cost, the number of iterations over the underlying DQNN. However, when using the global cost, the number of needed qudits also scales with the number of iterations over the underlying DQNN, making the local cost easier to compute on near-term quantum devices. Regardless of the cost, if we want to use the network to produce global DQRNN outputs, the number of needed qubits naturally scales with the number of iterations over the underlying DQNN.

We presented classical training algorithms for the local cost with pure and mixed product training data and the global cost with product or MPS training data and pure target outputs. Those classical algorithms can either be seen as a simulation of training DQRNNs on a quantum computer or as classically training DQRNNs on a classical representation of quantum states. The size of the matrices used in the training algorithm scale only with the width of the underlying DQNN and the bond dimension of the MPS training data, in case we use MPS and not product data. It explicitly does not scale with the number of iterations over the underlying DQNN or its depth and is efficient in that sense. However, if we want to calculate the global output of the DQRNN, like in the quantum algorithm, the size of the needed matrices scales with the number of iterations over the underlying DQNN.

The classical algorithms were numerically tested on tasks needing memory. We created the data with the delay channel, a standard example for a channel with memory, and the time evolution of a state under a time-dependent Hamiltonian. For these tasks, as expected, DQRNNs perform better than ff DQNNs on both the training and validation sets. For the delay channel, DQRNNs can generalize to unseen data with only a few training samples. Although both tasks need some form of memory, for the delay channel, quantum memory is needed, which is not apparent for learning the time evolution of a state w.r.t. a time-dependent Hamiltonian. In the latter task, classical memory or introducing a classical parameter into the unitaries of the ff DQNN might be enough. Future work could focus on categorizing tasks by the kind of memory needed or exploring DQNNs with additional classical parameters and comparing those to DQRNNs. Furthermore, we could investigate the relation between the presented DQRNN training algorithms and TN approaches for training classical NNs [69].

Many open questions remain regarding the trainability of DQRNNs. As shown in [248], when training deep fc ff DQNNs, we face Barren plateaus independent of whether the cost is local or global, which is not the case for shallow, sparse ff DQNNs. Those DQNNs only exhibit Barren plateaus if a global cost is used and not if a local cost is used. DQRNNs of small or fixed width can be seen as deep, sparse DQNNs sharing many parameters across layers. Although [248] does not consider this case, we numerically saw only a few small plateaus using the local cost but initially had problems with plateaus using the global cost. However, dividing the learning rate by the infidelity helps. One potential avenue for further investigation is to examine if this could be a general strategy for dealing with Barren plateaus in DQRNNs and DQNNs, at least when the DQNNs are shallow and sparse. Beside RNNs, a common way of dealing with sequential data is using a history of states as input of a ff NN, i.e., not only the input at time t is used but also inputs from previous time steps $t-n, ..., t$, which increases the dimensionality of the NN. Hence, a future direction of research could be to evaluate if using a DQRNN would lead to fewer trainability issues than using a history and a ff DQNN of higher dimension.

Another application of RNNs is found in environment models for RL or algorithms. In this line of thought, DQRNNs, together with measurements, could be used to learn an environment model when using model-based RL methods in quantum mechanical environments. The agent-environment interaction of a classical or hybrid agent and a quantum environment can be modeled with a quantum observable Markov decision process (QOMDP), which involves a quantum state space, an instrument and a reward operator for each action, and observations as the measurement results [362]. Combining DQRNNs with measurements on the output states, the DQRNN outputs could model observations and rewards, while the memory could model the environment state. This opens another line of research for DQNNs, training DQNNs using measurement results and using DQRNNs with measurements for model-based RL in quantum environments.

Generally, studying the use of RL in quantum mechanical environments is of interest to the number of control tasks faced in QI laboratories. One of the tasks often repeated in optics labs and, in particular, in QI labs is coupling a laser beam into an optical fiber. Although light is generally a quantum mechanical system, this task - at least in the standard mode - can be fully described by Gaussian beam optics. Nevertheless, we see it as a first step towards using RL for the control of quantum mechanical systems. We designed an RL environment for experimental fiber coupling and successfully trained several agents to perform fiber coupling in the lab.

The RL environment is designed as a partially observable Markov decision process (POMDP) with a goal. Each episode starts with a relatively low coupling efficiency, in particular, coupling efficiencies in the range of 10% to 80% coupling and medians around 25%. The agent can then perform continuous actions by moving four actuators to tilt two mirrors both in the horizontal and the vertical direction, which changes the spot and the angle with which the beam hits the fiber. Its observations are purely based on the coupling efficiency and past actions. If the actuator's positions are part of the observation, at least when only training with a specific optimal position, the agent can simply learn to find the optimal positions during training. However, if the experiment before the fiber is aligned differently, the optimal positions change, which renders the mentioned agent unusable for its specific use case. Hence, the actuator's positions are not part of the observation. The agent reaches its goal if the coupling efficiency reaches a specific goal power, which we varied between 75% and 91%, and fails if the coupling efficiency drops below 5%. It is rewarded based on the

coupling efficiency. Although our main aim is to train the agent directly in the experiment, we develop a simple virtual testbed for testing. Both the lab environment and the virtual testbed are implemented with Gymnasium [338].

Using the virtual testbed, we test different model-free algorithms and hyperparameters of the environment, which significantly reduces training time in the lab. As each training step in the lab takes about a second, we put a special focus on sample efficiency. For this limited amount of training time, algorithms using a replay buffer perform better than algorithms that do not, as expected. Soft actor-critic (SAC) performed best in the virtual testbed, closely followed by truncated quantile critics (TQC) and twin delayed deep deterministic policy gradient (TD3). Advantage actor-critic (A2C) performed worse, followed by proximal policy optimization (PPO) and deep deterministic policy gradient (DDPG). In particular, we tuned the following environment parameters: the maximum action, the reward parameters, the observation, and the maximum length of the episodes. We found that employing curriculum learning [343] by slowly increasing the goal power over time can be beneficial for high goal powers. Furthermore, although the design of the reset methods leads to consecutive episodes not being entirely independent, this has no major impact on the agents' performance.

In the laboratory, we used the parameters optimized in the virtual testbed and tested both TQC and SAC for a moderately high goal power. As TQC performed better, we used TQC for the other experiments in the lab. We showed how a TQC agent, over four days, successfully and reliably learns to couple the beam into the fiber with an efficiency of 90% without pre-training on the virtual testbed. For moderately high goal powers in the range of 85% to 87%, this training time is reduced to approximately 20 hours. In doing so, the agents learn to deal with the significant imprecision in the employed actuators. Furthermore, curriculum learning with lower goals and pre-training on the virtual testbed without noise can be helpful, although the first is more suitable and does not need time spent on developing a virtual testbed. However, an agent trained solely in the virtual testbed without noise still performs remarkably well in the lab environment. Although it is significantly slower than the agents trained in the lab, it still reaches a goal of 90% reliably in approximately the same amount of time as a human.

Future directions for using RL for fiber coupling include applying model-based RL, evaluating RL for fiber coupling of other modes of light, or combining RL with other methods to start the alignment without any light exiting the fiber. The task of fiber coupling can easily be scaled to more complex spatial alignment tasks by dividing systems into mirror-mirror-sensor blocks and solving them in sequence. Hence, we can use RL agents trained for fiber coupling on general spatial alignment problems.

A number of different alignment and control tasks in the optics lab could be automated using RL. Cavities are, for example, usually stabilized using the Pound-Drever-Hall procedure; see, e.g., [354] for an introduction. This procedure needs a number of additional components like phase modulators [355], homodyne detectors [356, 357], or split detectors [358, 359]. An RL agent could possibly simplify this setup by only using one additional sensor and no modulation and relying solely on the reflection or transmission power for decision-making. This would not only simplify the task of stabilizing cavities but pave the way to novel control strategies, such as for the phase control of squeezed vacuum states, whose current control strategies rely on introducing auxiliary laser fields [360] or unwanted noise [361]. RL could potentially perform these tasks without introducing additional laser fields or noise.

This thesis contributes to the goal of using ML for quantum control. Long-term research goals include the direct use of RL in quantum environments. Here, RL was successfully used in an optical environment that can still be modeled by Gaussian beam optics. One research branch is to employ RL for the control of more intricate quantum experiments directly in the lab. Another branch is to develop specialized RL methods for quantum environments, either using quantum, hybrid, or classical algorithms. Possibly, DQRNNs together with measurements can be used to model such environments and can hence influence this development. Ultimately, one could consider using ML methods for full quantum control like coherent control [363, 364] or coherent quantum noise cancellation [365].

A

Derivation of Training Algorithms for Different DQNN architectures

A.1 Feed-forward DQNN with Pure Output

This section is based on calculations performed in [244, 245], which are presented here to make the subsequent sections easier to understand.

Here, we are faced with a single feed-forward DQNN trained on a set of the form

$$S = \left(\left(\rho_1^{\text{in}}, \sigma_1^{\text{out}} \right), ..., \left(\rho_N^{\text{in}}, \sigma_N^{\text{out}} \right) \right)$$

where

$$\sigma_x^{\text{out}} = |\psi_x^{\text{out}}\rangle\langle\psi_x^{\text{out}}|.$$

We use the infidelity as our loss

$$l(\mathcal{N}_{\mathcal{U}}, (\rho^{\text{in}}, \sigma^{\text{out}})) = 1 - F(\mathcal{N}_{\mathcal{U}}(\rho^{\text{in}}), \sigma^{\text{out}})), \tag{A.1.1}$$

and hence the cost function

$$C_S = 1 - \frac{1}{N} \sum_{x=1}^{N} F(\rho_x^{\text{out}}, \sigma_x^{\text{out}}) = 1 - \sum_{x=1}^{N} \text{tr}(\rho_x^{\text{out}} \sigma_x^{\text{out}}) \tag{A.1.2}$$

with

$$\rho_x^{\text{out}} = \mathcal{N}_{\mathcal{U}}(\rho_x^{\text{in}}).$$

Notation wise, we use out and $L+1$ as well as in and 0 interchangeably.

A.1.1 Layer-to-Layer channels

We can write

$$\rho_x^{L+1} = \mathcal{N}_{\mathcal{U}}(\rho_x^0) = \mathcal{E}^{L+1}\left(\mathcal{E}^L\left(...\mathcal{E}^1\left(\rho_x^0\right)\right)\right) \tag{A.1.3}$$

where

$$\mathcal{E}^l(X^{l-1}) = \text{tr}^{l-1}\left(U^l\left(X^{l-1} \otimes |0...0\rangle^l \langle 0...0|\right) U^{l\dagger}\right) \tag{A.1.4}$$

as we will show later (see Equation (A.1.12)). Now, let us find the adjoint channel $\mathcal{E}^{l\dagger}$ of $\mathcal{E}^l$. Let $\{|\gamma\rangle\}$ be an orthonormal basis on the $l-1^{\text{th}}$ layer, $|\alpha\rangle$, $|\beta\rangle$ any vectors on that layer, and $|i\rangle$, $|j\rangle$ any vectors on the l^{th} layer. Then it is

$$\begin{aligned}
\operatorname{tr}\left(\left(\mathcal{E}^{l\dagger}\left(|i\rangle\langle j|\right)\right)^{\dagger}|\alpha\rangle\langle\beta|\right) &= \operatorname{tr}\left(|j\rangle\langle i|\,\mathcal{E}\left(|\alpha\rangle\langle\beta|\right)\right) = \langle i|\,\mathcal{E}_s^l\left(|\alpha\rangle\langle\beta|\right)|j\rangle \\
&= \left\langle i\right|\operatorname{tr}^{l-1}\left(U^l\left(|\alpha\rangle\langle\beta|\otimes|0\ldots0\rangle^l\langle0\ldots0|\right)U^{l\dagger}\right)\left|j\right\rangle \\
&= \sum_{\gamma}\left\langle\gamma,i\right|U^l\left(|\alpha\rangle\langle\beta|\otimes|0\ldots0\rangle^l\langle0\ldots0|\right)U^{l\dagger}\left|\gamma,j\right\rangle \\
&= \sum_{\gamma}\left\langle\gamma,i\right|U^l|\alpha,0\ldots0\rangle\langle\beta,0\ldots0|U^{l\dagger}|\gamma,j\rangle \\
&= \sum_{\gamma}\langle\beta,0\ldots0|U^{l\dagger}|\gamma,j\rangle\langle\gamma,i|\,U^l|\alpha,0\ldots0\rangle \\
&= \langle\beta,0\ldots0|U^{l\dagger}\left(\mathbb{1}^{l-1}\otimes|j\rangle\langle i|\right)U^l|\alpha,0\ldots0\rangle \\
&= \left\langle\beta\right|\operatorname{tr}^l\left(\left(\mathbb{1}^{l-1}\otimes|0\ldots0\rangle^l\langle0\ldots0|\right)U^{l\dagger}\left(\mathbb{1}^{l-1}\otimes|j\rangle\langle i|\right)U^l\right)\left|\alpha\right\rangle \\
&= \operatorname{tr}\Bigg(\operatorname{tr}^l\left(\left(\mathbb{1}^{l-1}\otimes|0\ldots0\rangle^l\langle0\ldots0|\right)U^{l\dagger}\left(\mathbb{1}^{l-1}\otimes|j\rangle\langle i|\right)U^l\right) \\
&\quad \left|\alpha\right\rangle\left\langle\beta\right|\Bigg).
\end{aligned}$$

Hence, we get

$$\mathcal{E}^{l\dagger}(X^l) = \operatorname{tr}^l\left(\left(\mathbb{1}^{l-1}\otimes|0\ldots0\rangle^l\langle0\ldots0|\right)U^{l\dagger}\left(\mathbb{1}^{l-1}\otimes X^l\right)U^l\right). \tag{A.1.5}$$

A.1.2 Feed-forward and feed-backward

We define a forward and backward motion by setting

$$\rho_x^0 = \rho_x^{\text{in}}, \tag{A.1.6}$$
$$\rho_x^l = \mathcal{E}^l\left(\rho_x^{l-1}\right), \quad l = 1, ..., L+1, \tag{A.1.7}$$
$$\sigma_x^{L+1} = \sigma_x^{\text{out}}, \tag{A.1.8}$$
$$\sigma_x^l = \mathcal{E}^{l+1^{\dagger}}\left(\sigma_x^{l+1}\right), \quad l = L, ..., 0. \tag{A.1.9}$$

Next, we want to write down non-recursive formulas for ρ_x^l and σ_x^l. To do so, we use that for $A \in \mathcal{B}(\mathcal{H}_A)$ and $B \in \mathcal{B}(\mathcal{H}_A \otimes \mathcal{H}_B)$ for some Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$ it is

$$\begin{aligned}
\operatorname{tr}_B((A\otimes\mathbb{1}_B)B) &= \sum_k(\mathbb{1}_A\otimes\langle k|)(A\otimes\mathbb{1}_B)B(\mathbb{1}_A\otimes|k\rangle) = \sum_k A(\mathbb{1}_A\otimes\langle k|)B(\mathbb{1}_A\otimes|k\rangle) \\
&= A\sum_k(\mathbb{1}_A\otimes\langle k|)B(\mathbb{1}_A\otimes|k\rangle) = A\operatorname{tr}_B(B).
\end{aligned} \tag{A.1.10}$$

By keeping in mind that $U^l = \mathbb{1}^{0:l-1} \otimes U^l \otimes \mathbb{1}^{l+2:L+1}$, where $i : j = i, ..., j$ for $i, j \in \mathbb{N}$, implicitly, and using this identity, we get

$$\begin{aligned}
\rho_x^l &= \mathcal{E}^l\left(\rho_x^{l-1}\right) = \mathcal{E}^l\left(\mathcal{E}^{l-1}\left(\rho_x^{l-2}\right)\right) \\
&= \mathrm{tr}^{l-1}\left(U^l\left(\mathrm{tr}^{l-2}\left(U^{l-1}\left(\rho_x^{l-2}\otimes|0...0\rangle^{l-1}\langle 0...0|\right)U^{l-1\dagger}\right)\otimes|0...0\rangle^{l}\langle 0...0|\right)U^{l\dagger}\right) \\
&= \mathrm{tr}^{l-1:l-2}\left(U^l\left(U^{l-1}\left(\rho_x^{l-2}\otimes|0...0\rangle^{l-1}\langle 0...0|\right)U^{l-1\dagger}\otimes|0...0\rangle^{l}\langle 0...0|\right)U^{l\dagger}\right) \\
&= \mathrm{tr}^{l-1:l-2}\left(U^l\left(U^{l-1}\left(\rho_x^{l-2}\otimes|0...0\rangle^{l-1}\langle 0...0|\right)\otimes\mathbb{1}^l\right)\left(U^{l-1\dagger}\otimes\mathbb{1}^l\right)\right. \\
&\quad \left.\left(\mathbb{1}^{l-2:l-1}\otimes|0...0\rangle^{l}\langle 0...0|\right)U^{l\dagger}\right) \\
&= \mathrm{tr}^{l-1:l-2}\left(U^l\left(U^{l-1}\left(\rho_x^{l-2}\otimes|0...0\rangle^{l-1}\langle 0...0|\right)\otimes\mathbb{1}^l\right)\left(\mathbb{1}^{l-2:l-1}\otimes|0...0\rangle^{l}\langle 0...0|\right)\right. \\
&\quad \left.\left(U^{l-1\dagger}\otimes\mathbb{1}^l\right)U^{l\dagger}\right) \\
&= \mathrm{tr}^{l-1:l-2}\left(U^lU^{l-1}\left(\rho_x^{l-2}\otimes|0...0\rangle^{l-1:l}\langle 0...0|\right)U^{l-1\dagger}U^{l\dagger}\right).
\end{aligned}$$

Doing the same thing recursively, we end up with

$$\rho_x^l = \mathcal{E}^l\left(\dots\mathcal{E}^1\left(\rho_x^0\right)\dots\right) \tag{A.1.11}$$

$$= \mathrm{tr}^{0:l-2}\Big(U^{1:l-1}\Big(\rho_x^0\otimes|0\dots0\rangle^{1:l-1}\langle 0\dots0|\Big)U^{1:l-1\dagger}\Big). \tag{A.1.12}$$

where we set $U^{l_1:l_2} = U^{l_2}\dots U^{l_1}$ for $l_1 \leq l_2$, and $U^{l_1:l_2} = \mathbb{1}$ for $l_1 > l_2$. For $l = L+1$, this proves Equation (4.5.4). In the same way, we can write

$$\begin{aligned}
\sigma_x^l &= \mathcal{E}^{l+1\dagger}\left(\mathcal{E}^{l+2\dagger}\left(\sigma_x^{l+2}\right)\right) \\
&= \mathrm{tr}^{l+1}\left(\left(\mathbb{1}^l\otimes|0\dots0\rangle^{l+1}\langle 0\dots0|\right)U^{l+1\dagger}\left(\mathbb{1}^l\otimes\mathrm{tr}^{l+2}\left(\left(\mathbb{1}^{l+1}\otimes|0\dots0\rangle^{l+2}\langle 0\dots0|\right)\right.\right.\right. \\
&\quad \left.\left.\left.U^{l+2\dagger}\left(\mathbb{1}^{l+1}\otimes\sigma_x^{l+2}\right)U^{l+2}\right)\right)U^{l+1}\right) \\
&= \mathrm{tr}^{l+1:l+2}\left(\left(\mathbb{1}^l\otimes|0\dots0\rangle^{l+1}\langle 0\dots0|\otimes\mathbb{1}^{l+2}\right)\left(U^{l+1\dagger}\otimes\mathbb{1}^{l+2}\right)\left(\mathbb{1}^{l:l+1}\right.\right. \\
&\quad \left.\left.\otimes|0\dots0\rangle^{l+2}\langle 0\dots0|\right)\left(\mathbb{1}^l\otimes U^{l+2\dagger}\right)\left(\mathbb{1}^{l:l+1}\otimes\sigma_x^{l+2}\right)\left(\mathbb{1}^l\otimes U^{l+2}\right)\left(U^{l+1}\otimes\mathbb{1}^{l+2}\right)\right) \\
&= \mathrm{tr}^{l+1:l+2}\left(\left(\mathbb{1}^l\otimes|0\dots0\rangle^{l+1}\langle 0\dots0|\otimes\mathbb{1}^{l+2}\right)\left(\mathbb{1}^{l:l+1}\otimes|0\dots0\rangle^{l+2}\langle 0\dots0|\right)\right. \\
&\quad \left.\left(U^{l+1\dagger}\otimes\mathbb{1}^{l+2}\right)\left(\mathbb{1}^l\otimes U^{l+2\dagger}\right)\left(\mathbb{1}^{l:l+1}\otimes\sigma_x^{l+2}\right)\left(\mathbb{1}^l\otimes U^{l+2}\right)\left(U^{l+1}\otimes\mathbb{1}^{l+2}\right)\right) \\
&= \mathrm{tr}^{l+1:l+2}\left(\left(\mathbb{1}^l\otimes|0\dots0\rangle^{l+1:l+2}\langle 0\dots0|\right)U^{l+1\dagger}U^{l+2\dagger}\left(\mathbb{1}^{l:l+1}\otimes\sigma_x^{l+2}\right)U^{l+2}U^{l+1}\right).
\end{aligned}$$

Doing this recursively, we end up with

$$\sigma_x^l = \mathcal{E}^{l+1\dagger}\left(\dots\mathcal{E}^{L+1\dagger}\left(\sigma_x^{\mathrm{out}}\right)\dots\right) \tag{A.1.13}$$

$$= \mathrm{tr}^{l+1:L+1}\left(\left(\mathbb{1}^l\otimes|0\dots0\rangle^{l+1:L+1}\langle 0\dots0|\right)U^{l+1:L+1\dagger}\left(\mathbb{1}^{l:L}\otimes\sigma_x^{\mathrm{out}}\right)U^{l+1:L+1}\right). \tag{A.1.14}$$

A.1.3 Change of Cost function

In each training step, we call the unitaries before the training step U_j^l. The unitaries are then updated according to

$$U_j^l \mapsto U_j^{l\prime} = e^{i\epsilon K_j^l}U_j^l \tag{A.1.15}$$

for some small number ϵ and a hermitian matrix K_j^l. In shorthand, we omit the dependence on U_j^l for most other variables depending on those unitaries and write them with a prime after the update. For example, during each training step, the cost changes from C to C', and we try to minimize the change in the cost function for small ϵ

$$\delta C = \lim_{\epsilon \to 0} \frac{C - C'}{\epsilon}. \tag{A.1.16}$$

To calculate δC, we first want to look at ρ_x^{out}. It is

$$\begin{aligned}
\rho_x^{L+1\,\prime} &= \mathcal{N}_{\mathcal{U}'}(\rho_x^0) \\
&= \text{tr}^{0:L}\Bigg(e^{i\epsilon K_{m_{L+1}}^{L+1}} U_{m_{L+1}}^{L+1} \dots e^{i\epsilon K_1^1} U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) \\
&\quad U_1^{1\,\dagger} e^{-i\epsilon K_1^1} \dots U_{m_{L+1}}^{L+1\ \dagger} e^{-i\epsilon K_{m_{L+1}}^{L+1}} \Bigg).
\end{aligned}$$

By Taylor expansion of the matrix exponential, we get

$$\begin{aligned}
\rho_x^{L+1\,\prime} = \text{tr}^{0:L}\Bigg(&\left(\mathbb{1}^{L+1} + i\epsilon K_{m_{L+1}}^{L+1}\right) U_{m_{L+1}}^{L+1} \dots \left(\mathbb{1}^1 + i\epsilon K_1^1\right) U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) \\
&U_1^{1\,\dagger} \left(\mathbb{1}^1 - i\epsilon K_1^1\right) \dots U_{m_{L+1}}^{L+1\ \dagger} \left(\mathbb{1}^{L+1} - i\epsilon K_{m_{L+1}}^{L+1}\right) \Bigg).
\end{aligned}$$

Only writing down the second-order terms, it is

$$\begin{aligned}
&= \text{tr}^{0:L}\left(U_{m_{L+1}}^{L+1} \dots U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) U_1^{1\,\dagger} \dots U_{m_{L+1}}^{L+1,\,\dagger} \right) + i\epsilon \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \\
&\quad \text{tr}^{0:L}\Bigg(U_{m_{L+1}}^{L+1} \dots U_{j-1}^l K_j^l U_j^l \dots U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) U_1^{1\,\dagger} \dots U_{m_{L+1}}^{L+1\ \dagger} \\
&\quad - U_{m_{L+1}}^{L+1} \dots U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) U_1^{1\,\dagger} \dots U_j^{l\,\dagger} K_j^l U_{j+1}^{l\,\dagger} \dots U_{m_{L+1}}^{L+1,\,\dagger} \Bigg) + \mathcal{O}(\epsilon^2) \\
&= \rho_x^{L+1} + i\epsilon \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \text{tr}^{0:L}\Bigg(U_{m_{L+1}}^{L+1} \dots U_{j-1}^l \Big[K_j^l, U_j^l \dots U_1^1 \left(\rho_x^0 \otimes |0\dots0\rangle_{1:L+1}\langle 0\dots 0|\right) \\
&\quad U_1^{1\,\dagger} \dots U_j^{l\,\dagger} \Big] U_{j+1}^{l\,\dagger} \dots U_{m_{L+1}}^{L+1,\,\dagger} \Bigg) + \mathcal{O}(\epsilon^2) \\
&= \rho_x^{L+1} + \epsilon \delta\rho_x^{L+1} + \mathcal{O}(\epsilon^2).
\end{aligned}$$

with

$$\delta\rho_x^{L+1} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \left(\rho_x'^{\,L+1} - \rho_x^{L+1} \right) \tag{A.1.17}$$

$$\begin{aligned}
&= i \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \text{tr}^{0:L}\Bigg(U^{l+1:L+1} U_{j+1:m_l}^l \Big[K_j^l, U_{1:j}^l U^{1:l-1} \left(\rho_x^0 \otimes |0\dots0\rangle^{1:L+1}\langle 0\dots 0|\right) \\
&\quad U^{1:l-1\,\dagger} U_{1:j}^{l\ \dagger} \Big] U_{j+1:m_l}^{l\ \dagger} U^{l+1:L+1\,\dagger} \Bigg)
\end{aligned} \tag{A.1.18}$$

where we write $U^l_{j_1:j_2} = U^l_{j_2} \dots U^l_{j_1}$ for $j_1 \leq j_2$, and $U^l_{j_1:j_2} = \mathbb{1}$ for $j_1 > j_2$.

Furthermore, it is

$$\begin{aligned} C' &= 1 - \frac{1}{N}\sum_{x=1}^{N} \left(\langle \phi_x^{L+1} | \, \rho_x^{L+1} \, | \phi_x^{L+1} \rangle + \epsilon \, \langle \phi_x^{L+1} | \, \delta\rho_x^{L+1} \, | \phi_x^{L+1} \rangle \right) + \mathcal{O}(\epsilon^2) \\ &= C - \epsilon \frac{1}{N}\sum_{x=1}^{N} \langle \phi_x^{L+1} | \, \delta\rho_x^{L+1} \, | \phi_x^{L+1} \rangle + \mathcal{O}(\epsilon^2) = C - \epsilon \frac{1}{N}\sum_{x=1}^{N} \mathrm{tr}\left(\sigma_x^{L+1} \delta\rho_x^{L+1} \right) + \mathcal{O}(\epsilon^2) \end{aligned}$$

and hence,

$$\begin{aligned} \delta C = & -\lim_{\epsilon \to 0} \frac{C - C'}{\epsilon} = -\frac{1}{N}\sum_{x=1}^{N} \mathrm{tr}\left(\sigma_x^{L+1} \delta\rho_x^{L+1} \right) && \text{(A.1.19)} \\ = & -\frac{1}{N}\sum_{x=1}^{N} i \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \mathrm{tr}\Big(\sigma_x^{L+1} \mathrm{tr}^{0:L}\Big(U^{l+1:L+1} U^l_{j+1:m_l} \Big[K^l_j, \\ & U^l_{1:j} U^{1:l-1} \left(\rho_x^0 \otimes |0 \dots 0\rangle^{1:L+1} \langle 0 \dots 0| \right) U^{1:l-1\dagger} U^{l\dagger}_{1:j} \Big] U^l_{j+1:m_l}{}^{\dagger} U^{l+1:L+1\dagger} \Big) \Big) \\ = & -\frac{i}{N}\sum_{x=1}^{N} \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \mathrm{tr}\Big(\left(\mathbb{1}^{0:L} \otimes \sigma_x^{L+1} \right) U^{l+1:L+1} U^l_{j+1:m_l} \Big[K^l_j, \\ & U^l_{1:j} U^{1:l-1} \left(\rho_x^0 \otimes |0 \dots 0\rangle^{1:L+1} \langle 0 \dots 0| \right) U^{1:l-1\dagger} U^{l\dagger}_{1:j} \Big] U^l_{j+1:m_l}{}^{\dagger} U^{l+1:L+1\dagger} \Big) \\ = & -\frac{i}{N}\sum_{x=1}^{N} \sum_{l=1}^{L+1} \sum_{j=1}^{m_l} \mathrm{tr}\Big(U^{l+1:L+1\dagger} \left(\mathbb{1}^{0:L} \otimes \sigma_x^{L+1} \right) U^{l+1:L+1} \\ & U^l_{j+1:m_l} \Big[K^l_j, U^l_{1:j} U^{1:l-1} \left(\rho_x^0 \otimes |0 \dots 0\rangle^{1:L+1} \langle 0 \dots 0| \right) U^{1:l-1\dagger} U^{l\dagger}_{1:j} \Big] U^l_{j+1:m_l}{}^{\dagger} \Big). && \text{(A.1.20)} \end{aligned}$$

With

$$A = U^{l+1:L+1\dagger} \left(\mathbb{1}^{0:L} \otimes \sigma_x^{L+1} \right) U^{l+1:L+1} \qquad \text{(A.1.21)}$$

$$B = U^l_{j+1:m_l} \Big[K^l_j, U^l_{1:j} U^{1:l-1} \left(\rho_x^0 \otimes |0 \dots 0\rangle^{1:L+1} \langle 0 \dots 0| \right) U^{1:l-1\dagger} U^{l\dagger}_{1:j} \Big] U^l_{j+1:m_l}{}^{\dagger} \qquad \text{(A.1.22)}$$

it is

$$\begin{aligned}
\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\left(\left(\mathbb{1}^{0:l-1}\otimes A\right)\left(B\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)\right)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\left(A\left(\operatorname{tr}^{0:l-1}(B)\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)\right)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\left(A\left(\operatorname{tr}^{0:l-1}(B)\otimes\mathbb{1}^{l+1:L+1}\right)\left(\mathbb{1}_x^l\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)\right)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\left(\left(\mathbb{1}_x^l\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)A\left(\operatorname{tr}^{0:l-1}(B)\otimes\mathbb{1}_x^{l+1:L+1}\right)\right)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\left(\operatorname{tr}^{l+1:L+1}\left(\left(\mathbb{1}_x^l\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)A\right)\operatorname{tr}^{0:l-1}(B_x)\right).
\end{aligned}\tag{A.1.23}$$

These partial traces now have to be expressed in terms of the earlier discussed states arising from the feed-forward and feed-backward motion. With (A.1.22) it follows that

$$\begin{aligned}
&\operatorname{tr}^{0:l-1}(B)\\
&\quad=\operatorname{tr}^{0:l-1}\left(U_{j+1:m_l}^l\left[K_j^l,U_{1:j}^lU^{1:l-1}\left(\rho_x^0\otimes|0\ldots0\rangle^{1:L+1}\langle 0\ldots0|\right)U^{1:l-1\dagger}U_{1:j}^{l\ \dagger}\right]U_{j+1:m_l}^{l\ \dagger}\right)\\
&\quad=\operatorname{tr}^{l-1}\Big(U_{j+1:m_l}^l\Big[K_j^l,U_{1:j}^l\Big(\operatorname{tr}^{0:l-2}\Big(U^{1:l-1}\Big(\rho_x^0\otimes|0\ldots0\rangle^{1:l-1}\langle 0\ldots0|\Big)\\
&\qquad U^{1:l-1\dagger}\Big)\otimes|0\ldots0\rangle^l\langle 0\ldots0|\Big)U_{1:j}^{l\ \dagger}\Big]U_{j+1:m_l}^{l\ \dagger}\Big).
\end{aligned}$$

Using Equation (A.1.12), we get

$$\operatorname{tr}^{0:l-1}(B)=\operatorname{tr}^{l-1}\Big(U_{j+1:m_l}^l\Big[K_j^l,U_{1:j}^l\Big(\rho_x^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|\Big)U_{1:j}^{l\ \dagger}\Big]U_{j+1:m_l}^{l\ \dagger}\Big).\tag{A.1.24}$$

By using Equations (A.1.21) and (A.1.14), we have

$$\begin{aligned}
&\operatorname{tr}^{l+1:L+1}\left(\left(\mathbb{1}^l\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)A\right)\\
&=\operatorname{tr}^{l+1:L+1}\left(\left(\mathbb{1}^l\otimes|0\ldots0\rangle^{l+1:L+1}\langle 0\ldots0|\right)U^{l+1:L+1\dagger}\left(\mathbb{1}^{l\ldots L}\otimes\sigma_x^{L+1}\right)U^{l+1:L+1}\right)\\
&=\sigma_x^l.
\end{aligned}$$

Together, we get

$$\begin{aligned}
\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\sigma_x^l\operatorname{tr}^{l-1}\Big(U_{j+1:m_l}^l\Big[K_j^l,U_{1:j}^l\Big(\rho_x^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|\Big)U_{1:j}^{l\ \dagger}\Big]\\
&U_{j+1:m_l}^{l\ \dagger}\Big)\Big)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\left(\mathbb{1}^{l-1}\otimes\sigma_x^l\right)U_{j+1:m_l}^l\Big[K_j^l,U_{1:j}^l\Big(\rho_x^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|\Big)U_{1:j}^{l\ \dagger}\Big]\\
&U_{j+1:m_l}^{l\ \dagger}\Big).
\end{aligned}$$

As it is

$$\operatorname{tr}(A[B,C]D) = \operatorname{tr}(ABCD - ACBD) = \operatorname{tr}(CDAB - DACB) = \operatorname{tr}([C,DA]B) \quad \text{(A.1.25)}$$

for all $A, B, C, D \in \mathcal{B}(\mathcal{H})$ for some Hilbert space $\mathcal{H}$, we have

$$\begin{aligned}\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\Big[U_{1:j}^{l}\Big(\rho_x^{l-1}\otimes|0\ldots0\rangle^{l}\langle 0\ldots0|\Big)U_{1:j}^{l}{}^{\dagger}, U_{j+1:m_l}^{l}{}^{\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_x^{l}\Big)\\ & U_{j+1:m_l}^{l}\Big]K_j^{l}\Big)\\ = & -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(M_j^{l}K_j^{l}\Big)\end{aligned} \quad \text{(A.1.26)}$$

with

$$M_j^{l} = \frac{1}{N}\sum_{x=1}^{N}\Big[U_{1:j}^{l}\Big(\rho_x^{l-1}\otimes|0\ldots0\rangle^{l}\langle 0\ldots0|\Big)U_{1:j}^{l}{}^{\dagger}, U_{j+1:m_l}^{l}{}^{\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_x^{l}\Big)U_{j+1:m_l}^{l}\Big]. \quad \text{(A.1.27)}$$

A.1.4 Update Matrices and Proof of Proposition 4.2

Now, to get our update matrices K_j^l we only have to minimise

$$\delta C = -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(M_j^{l}K_j^{l}\Big).$$

To do so, write

$$K_j^{l} = \sum_{\alpha_1,\alpha_2,\ldots,\alpha_{m_{l-1}},\beta} K^{l}_{j,\alpha_1,\ldots,\alpha_{m_{l-1}},\beta}\left(\sigma^{\alpha_1}\otimes\ \ldots\ \otimes\sigma^{\alpha_{m_{l-1}}}\otimes\sigma^{\beta}\right).$$

Since δC is linear in the $K^{l}_{j,\alpha_1,\ldots,\alpha_{m_{l-1}}}$, the maximum is reached for $\pm\infty$. But we only want to make small (finite) steps in the direction that maximises δC, so we impose the constraint

$$\sum_{\alpha_i,\beta=0}^{1}\left(K^{l}_{j,\alpha_i,\beta}\right)^2 = c = \text{const.}$$

By introducing a Lagrange multiplier $\lambda \in \mathbb{R}$, we get the optimization problem

$$\begin{aligned}\min_{K^{l}_{j,\alpha_1 i,\beta}} & -i\sum_{l'=1}^{L+1}\sum_{j'=1}^{m_{l'}}\sum_{\alpha_i',\beta'=0}^{1}\operatorname{tr}\Big(M_{j'}^{l'}K^{l'}_{j',\alpha_i',\beta'}\left(\sigma^{\alpha_1'}\otimes\ \ldots\ \otimes\sigma^{\alpha'_{m_{l'-1}}}\otimes\sigma^{\beta'}\right)\Big)\\ & +\lambda\left(\sum_{l'=1}^{L+1}\sum_{j'=1}^{m_{l'}}\sum_{\alpha_i',\beta'=0}^{1}\left(K^{l'}_{j',\alpha_i',\beta'}\right)^2 - c\right).\end{aligned}$$

By setting the derivative w.r.t. $K^{l}_{j,\alpha_i,\beta}$ equal to 0, we get

$$-i\operatorname{tr}\Big(M_j^{l}\left(\sigma^{\alpha_1}\otimes\ \ldots\ \otimes\sigma^{\alpha_{m_{l-1}}}\otimes\sigma^{\beta}\right)\Big) + 2\lambda K^{l}_{j,\alpha_i,\beta} = 0$$

which is equivalent to

$$K^l_{j,\alpha_i,\beta} = \frac{i}{2\lambda}\mathrm{tr}\Big(M^l_j\left(\sigma^{\alpha_1}\otimes\ldots\otimes\sigma^{\alpha_{m_{l-1}}}\otimes\sigma^{\beta}\right)\Big).$$

Hence, it is

$$K^l_j = \sum_{\alpha_1,\alpha_2,\ldots,\alpha_{m_{l-1}},\beta}\frac{i}{2\lambda}\mathrm{tr}\Big(M^l_j\left(\sigma^{\alpha_1}\otimes\ldots\otimes\sigma^{\alpha_{m_{l-1}}}\otimes\sigma^{\beta}\right)\Big)\left(\sigma^{\alpha_1}\otimes\ \ldots\ \otimes\sigma^{\alpha_{m_{l-1}}}\otimes\sigma^{\beta}\right).$$

Using the completeness of Pauli matrices, we get

$$K^l_j = \frac{i\cdot 2^{m_{l-1}+1}}{2\lambda}\mathrm{tr}^l_{1:j-1}\Big(\mathrm{tr}^l_{j+1:m_l}\left(M^l_j\right)\Big) = \frac{2^{m_{l-1}}i}{\lambda}\mathrm{tr}^l_{1:j-1}\Big(\mathrm{tr}^l_{j+1:m_l}\left(M^l_j\right)\Big)$$

which concludes the proof of Proposition 4.2. Hence, in each step, we perform the update

$$U^l_j \mapsto \exp\left(P^l_j\right)U^l_j$$

with

$$P^l_j = i\epsilon K^l_j = i\epsilon\frac{2^{m_{l-1}}i}{\lambda}\mathrm{tr}^l_{1:j-1}\Big(\mathrm{tr}^l_{j+1:m_l}\left(M^l_j\right)\Big) = -2^{m_{l-1}}\eta\,\mathrm{tr}^l_{1:j-1}\Big(\mathrm{tr}^l_{j+1:m_l}\left(M^l_j\right)\Big),$$

where $\eta = \frac{\epsilon}{\lambda}$ is the so called learning rate.

A.1.5 The algorithm

In total, we get Algorithm 9.

A.2 Feed-forward DQNN with Mixed Output

The derivation presented in this section is based on work with Nils Renziehausen in the context of his Bachelor's thesis [252] and is presented here to make the next sections easier to understand.
Again, we are faced with a single feed-forward DQNN trained on a set of the form

$$S = \left(\left(\rho^{\mathrm{in}}_1,\sigma^{\mathrm{out}}_1\right),\ldots,\left(\rho^{\mathrm{in}}_N,\sigma^{\mathrm{out}}_N\right)\right)\in\left(\mathcal{D}(\mathcal{H}^{\mathrm{in}})\times\mathcal{D}(\mathcal{H}^{\mathrm{out}})\right)^{\times N}$$

but now, in general there is no $\psi^{\mathrm{out}}_x\in\mathcal{H}^{\mathrm{out}}$ with $\sigma^{\mathrm{out}}_x = |\psi^{\mathrm{out}}_x\rangle\langle\psi^{\mathrm{out}}_x|$. We use the cost function

$$C_S = \frac{1}{N}\sum_{x=1}^{N}\mathrm{tr}((\rho^{\mathrm{out}}_x-\sigma^{\mathrm{out}}_x)^2) \tag{A.2.1}$$

and know

$$\rho^{\mathrm{out}}_x = \mathcal{N}_{\mathcal{U}}(\rho^{\mathrm{in}}_x).$$

Again, we use out and $L+1$ as well as in and 0 interchangeably in terms of notation.

Algorithm 14 Classical Training Algorithm of ff DQNNs with pure target [245]

Input: $((\rho_x^{\text{in}}, |\psi_x^{\text{out}}\rangle))_{x=1,\dots,N}$, $(m_l)_{l=0,\dots,L+1}$ or $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U_j^l)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      $U_j^l \leftarrow$RandomUnitaryHaar$(m_{l-1}+1$ qudits$)$ ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits*

      $U_j^l \leftarrow \mathbb{1}_{1:j-1}^l \otimes U_j^l \otimes \mathbb{1}_{j+1:m_l}^l$

for t=1,...,T **do**

  for x=1,...,N **do** ▷ *Feed-forward*

    $\rho_x^0 \leftarrow \rho_x^{\text{in}}$

    for l=1,...,L+1 **do**

      $\rho_x^l \leftarrow \text{tr}^{l-1}\left(U^l\left(\rho_x^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0|\right)U^{l\dagger}\right)$

  for x=1,...,N **do** ▷ *Feed-backward*

    $\sigma_x^{L+1} \leftarrow |\psi_x^{\text{out}}\rangle \langle\psi_x^{\text{out}}|$

    for l=0,...,L **do**

      $\sigma_x^l \leftarrow \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right)U^{l+1\dagger}\left(\mathbb{1}^l \otimes \sigma_x^{l+1}\right)U^{l+1}\right)$

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      $M_j^l \leftarrow \frac{1}{N}\sum_{x=1}^{N}\left[U_{1:j}^l\left(\rho_x^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0|\right)U_{1:j}^{l\dagger}, U_{j+1:m_l}^{l\dagger}\left(\mathbb{1}^{l-1} \otimes \sigma_x^l\right)U_{j+1:m_l}^l\right]$

      $P_j^l \leftarrow -2^{m_{l-1}}\eta\,\text{tr}_{1:j-1}^l\left(\text{tr}_{j+1:m_l}^l\left(M_j^l\right)\right)$

      $U_j^l \leftarrow \exp\left(P_j^l\right)U_j^l$

  $C_t \leftarrow 1 - \frac{1}{N}\sum_{x=1}^{N}\text{tr}\left(\rho_x\sigma_x\right)$

A.2.1 Derivative of Cost Function

In each training step, we again update our unitaries U_j^l according to

$$U_j^l \mapsto U_j^{l\prime} = e^{i\epsilon K_j^l}U_j^l \tag{A.2.2}$$

for some small number ϵ and a hermitian matrix K_j^l. First, we again have to find the derivative of the cost function. To do so, first rewrite the cost using the cylic rule of trace as

$$C = \frac{1}{N}\sum_{x=1}^{N}\text{tr}((\rho_x^{\text{out}})^2 + (\sigma_x^{\text{out}})^2 - 2\rho_x^{\text{out}}\sigma_x^{\text{out}}).$$

As σ_x^{out} is not changed by updating, it is

$$\begin{aligned}
C' &= \frac{1}{N}\sum_{x=1}^{N} \text{tr}((\rho'^{\text{out}}_x)^2 + (\sigma_x^{\text{out}})^2 - 2\rho'^{\text{out}}_x \sigma_x^{\text{out}}) \\
&= \frac{1}{N}\sum_{x=1}^{N} \text{tr}((\rho_x^{L+1} + \epsilon\delta\rho_x^{L+1} + \mathcal{O}(\epsilon^2))^2 + (\sigma_x^{\text{out}})^2 - 2(\rho_x^{L+1} + \epsilon\delta\rho_x^{L+1} + \mathcal{O}(\epsilon^2))\sigma_x^{\text{out}}) \\
&= \frac{1}{N}\sum_{x=1}^{N} \text{tr}((\rho_x^{L+1})^2 + 2\epsilon(\delta\rho_x^{L+1})\rho_x^{L+1} + (\sigma_x^{\text{out}})^2 - 2(\rho_x^{L+1} + \epsilon(\delta\rho_x^{L+1}))\sigma_x^{\text{out}}) + \mathcal{O}(\epsilon^2) \\
&= C + 2\epsilon\text{tr}((\delta\rho_x^{L+1})(\rho_x^{L+1} - \sigma_x^{\text{out}})) + \mathcal{O}(\epsilon^2),
\end{aligned}$$

so

$$\delta C = 2\text{tr}((\delta\rho_x^{L+1})(\rho_x^{L+1} - \sigma_x^{\text{out}})). \tag{A.2.3}$$

This is nearly the same as for pure outputs (Equation (A.1.19)). We only have $\sigma_x^{\text{out}} - \rho_x^{L+1}$ instead of σ_x^{out} and a factor of 2 that we can incorporate in the learning rate.

A.2.2 The algorithm

Hence, we only have to change $\sigma_x^{L+1} = \sigma_x^{\text{out}}$ to $\sigma_x^{L+1} = \sigma_x^{\text{out}} - \rho_x^{L+1}$ in Algorithm 14 and get Algorithm 15. Note that this means that σ_x^l is not a state anymore but is closer to the error used in classical NNs.

A.3 DQRNN with Local Cost for Separable Input and Output

If you are not familiar with the derivation of the classical training algorithms for DQNNs, we would advise you to read Section A.1 first. The derivation of this algorithm was first presented in [246]. Now, let us look at a DQRNN trained with the local cost, separable inputs, and, for now, pure, separable outputs. Given $N_\alpha,\ M \in \mathbb{N}$ for $\alpha = 1, ..., M$, let our training set be of the form

$$S = \left(\rho_{0\,\alpha}^{\text{mem}}, \left(\rho_{x\,\alpha}^{\text{in}}, \sigma_{x\,\alpha}^{\text{out}}\right)_{x=1,\dots,N_\alpha}\right)_{\alpha=1,\dots,M}$$

with

$$\sigma_{x\,\alpha}^{\text{out}} = |\psi_{x\,\alpha}^{\text{out}}\rangle\langle\psi_{x\,\alpha}^{\text{out}}|, \quad \psi_{x\,\alpha}^{\text{out}} \in \mathcal{H}^{\text{out}}, \quad \rho_{0\,\alpha}^{\text{mem}} \in \mathcal{D}(\mathcal{H}^{\text{mem}}), \quad \rho_{x\,\alpha}^{\text{in}} \in \mathcal{D}(\mathcal{H}^{\text{in}}).$$

If $\rho_{0\,\alpha}^{\text{mem}}$ is not given, we will set $\rho_{0\,\alpha}^{\text{mem}} = |0...0\rangle^{\text{mem}}\langle 0...0|$.

The local cost function for a DQRNN with pure outputs is given by the local infidelity on the outputs

$$C_S = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N} F(\rho_{x\,\alpha}^{\text{out}}, \sigma_{x\alpha}^{\text{out}}) = 1 - \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N}\text{tr}(\rho_{x\,\alpha}^{\text{out}}\sigma_{x\alpha}^{\text{out}}) \tag{A.3.1}$$

with

$$\rho_\alpha^{\text{IN}} := \rho_{0\,\alpha}^{\text{mem}} \otimes \rho_{1\,\alpha}^{\text{in}} \otimes \cdots \otimes \rho_{N_\alpha\,\alpha}^{\text{in}} \tag{A.3.2}$$

$$\rho_\alpha^{\text{OUT}} := \left(\left(\mathbb{1}_{1,\dots,N_\alpha-1}^{\text{out}} \otimes \mathcal{N}_\mathcal{U}\right) \circ \cdots \circ \left(\mathcal{N}_\mathcal{U} \otimes \mathbb{1}_{2,\dots,N_\alpha}^{\text{in}}\right)\right)\left(\rho^{\text{IN}}\right) =: \mathcal{M}_\mathcal{U}\left(\rho_\alpha^{\text{IN}}\right) \tag{A.3.3}$$

$$\rho_{x\,\alpha}^{\text{out}} := \text{tr}_{0:N_\alpha}^{0:L,\text{mem}}\left(\text{tr}_{0:x-1,x+1:N_\alpha}^{\text{out}}\left(\rho_\alpha^{\text{OUT}}\right)\right). \tag{A.3.4}$$

Algorithm 15 Classical Training Algorithm of ff DQNNs with mixed target [252]

Input: $((\rho_x^{\text{in}}, \sigma_x^{\text{out}}))_{x=1,\dots,N}$, $(m_l)_{l=0,\dots,L+1}$ or $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U_j^l)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      $U_j^l \leftarrow$RandomUnitaryHaar$(m_{l-1}+1$ qudits$)$ ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits*

      $U_j^l \leftarrow \mathbb{1}_{1:j-1}^l \otimes U_j^l \otimes \mathbb{1}_{j+1:m_l}^l$

for t=1,...,T **do**

  for x=1,...,N **do** ▷ *Feed-forward*

    $\rho_x^0 \leftarrow \rho_x^{\text{in}}$

    for l=1,...,L+1 **do**

      $\rho_x^l \leftarrow \text{tr}^{l-1}\left(U^l\left(\rho_x^{l-1} \otimes |0\dots0\rangle^l \langle 0\dots0|\right)U^{l\dagger}\right)$

  for x=1,...,N **do** ▷ *Backpropagation*

    $\sigma_x^{L+1} \leftarrow (\sigma_x^{\text{out}} - \rho_x^{L+1})$

    for l=0,...,L **do**

      $\sigma_x^l \leftarrow \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right)U^{l+1\dagger}\left(\mathbb{1}^l \otimes \sigma_x^{l+1}\right)U^{l+1}\right)$

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      $M_j^l \leftarrow \frac{1}{N}\sum_{x=1}^N \left[U_{1:j}^l\left(\rho_x^{l-1}\otimes|0\dots0\rangle^l\langle 0\dots0|\right)U_{1:j}^{l\dagger}, U_{j+1:m_l}^{l\dagger}\left(\mathbb{1}^{l-1}\otimes\sigma_x^l\right)U_{j+1:m_l}^l\right]$

      $P_j^l \leftarrow -2^{m_{l-1}+1}\eta\,\text{tr}_{1:j-1}^l\left(\text{tr}_{j+1:m_l}^l\left(M_j^l\right)\right)$

      $U_j^l \leftarrow \exp\left(P_j^l\right)U_j^l$

  $C_t \leftarrow \frac{1}{N}\sum_{x=1}^N \text{tr}\left(\left(\rho_x^{L+1} - \sigma_x^{\text{out}}\right)^2\right)$ ▷ *Calculate Cost*

The Hilbert spaces are related as $\mathcal{H}^0 = \mathcal{H}^{\text{mem}} \otimes \mathcal{H}^{\text{in}}$, $\mathcal{H}^{L+1} = \mathcal{H}^{\text{out}} \otimes \mathcal{H}^{\text{mem}}$, $\mathcal{H}^{\text{IN}} = \mathcal{H}^{\text{mem}} \otimes \mathcal{H}^{\text{in}\otimes N_\alpha}$ and $\mathcal{H}^{\text{OUT}} = \mathcal{H}^{\text{out}\otimes N_\alpha} \otimes \mathcal{H}^{\text{mem}}$. To keep things simple, we set $M = 1$ in the appendix. To get the case $M > 1$, we just have to average over the αs.

A.3.1 Network-to-Network Channels

We will use the same layer-to-layer channels as for the ff DQNN, but also introduce network channels: As introduced before, we have the network-to-network channel

$$\begin{aligned}\mathcal{N}_{\mathcal{U}} : \mathcal{H}^0 &\to \mathcal{H}^{L+1}\\ X^0 &\mapsto \text{tr}^{0:L}\left(\mathcal{U}\left(X^0 \otimes |0...0\rangle^{1:L+1}\langle 0...0|\right)\mathcal{U}^\dagger\right)\end{aligned}$$

and we know

$$\mathcal{N}_{\mathcal{U}} = \mathcal{E}^{L+1} \circ \dots \circ \mathcal{E}^1.$$

It's adjoint is then given by

$$\mathcal{N}_{\mathcal{U}}^\dagger = \mathcal{E}^{1\dagger} \circ \dots \circ \mathcal{E}^{L+1\dagger},$$

i.e.

$$\begin{aligned}\mathcal{N}_{\mathcal{U}}^{\dagger} &: \mathcal{H}^{L+1} \to \mathcal{H}^{0} \\ &X^{L+1} \mapsto \mathrm{tr}^{1:L+1}\left(\left(\mathbb{1}^{0} \otimes |0\ldots0\rangle^{1:L+1}\langle 0\ldots0|\right)\mathcal{U}^{\dagger}\left(\mathbb{1}^{0:L} \otimes X^{L+1}\right)\mathcal{U}\right).\end{aligned}$$

A.3.2 Feed-forward

We are looking at the local cost here and are only interested in the $\rho_{x\,\alpha}^{\text{out}}$, where we trace out everything except for the x^{th} output. As we will see later, we can either always carry the full state or trace out the memory and output after each network to get a local output ρ_x^{out} and memory ρ_x^{mem} state, respectively. Then, the local memory state is tensored to the next input ρ_x^{in}. This is then used as input for the next DQNN.

Let

$$\rho_x^0 = \rho_{x-1}^{\text{mem}} \otimes \rho_x^{\text{in}} \tag{A.3.5}$$

$$\rho_x^l = \mathcal{E}^l\left(\rho_x^{l-1}\right), \quad l = 1, ..., L+1 \tag{A.3.6}$$

$$\rho_x^{\text{mem}} = \mathrm{tr}^{\text{out}}\left(\rho_x^{L+1}\right) \tag{A.3.7}$$

$$\tilde{\rho}_x^{\text{out}} = \mathrm{tr}^{\text{mem}}\left(\rho_x^{L+1}\right). \tag{A.3.8}$$

As the layer-to-layer propagation is defined in the same way as for the ff DQNN, we have (see Equation (A.1.12))

$$\begin{aligned}\rho_x^l &= \mathrm{tr}^{0:l-1}\Big(U^{1:l}\Big(\rho_x^0 \otimes |0\ldots0\rangle^{1:l}\langle 0\ldots0|\Big)U^{1:l\,\dagger}\Big) \\ &= \mathrm{tr}^{0:l-1}\Big(U^{1:l}\Big(\rho_{x-1}^{\text{mem}} \otimes \rho_x^{\text{in}} \otimes |0\ldots0\rangle^{1:l}\langle 0\ldots0|\Big)U^{1:l\,\dagger}\Big).\end{aligned} \tag{A.3.9}$$

Together with Equation (A.3.7), this means, it is

$$\begin{aligned}\rho_x^{\text{mem}} &= \mathrm{tr}_x^{\text{out}}\left(U^{1:L+1}\left(\rho_x^{\text{in}} \otimes \rho_{x-1}^{\text{mem}} \otimes |0\ldots0\rangle_x^{1:L+1}\langle 0\ldots0|\right)U^{1:L+1\,\dagger}\right) \\ &= \mathrm{tr}_x^{0:L,\text{out}}\left(\mathcal{U}_x\left(\rho_x^{\text{in}} \otimes \rho_{x-1}^{\text{mem}} \otimes |0\ldots0\rangle_x^{1:L+1}\langle 0\ldots0|\right)\mathcal{U}_x^{\dagger}\right) \\ &= \mathrm{tr}_x^{0:L,\text{out}}\Big(\mathcal{U}_x\Big(\rho_x^{\text{in}} \otimes \mathrm{tr}_{x-1}^{0:L,\text{out}}\Big(\mathcal{U}_{x-1}\Big(\rho_{x-1}^{\text{in}} \otimes \rho_{x-2}^{\text{mem}} \otimes |0\ldots0\rangle_{x-1}^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_{x-1}^{\dagger}\Big) \\ &\quad \otimes |0\ldots0\rangle_x^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_x^{\dagger}\Big)\end{aligned}$$

where we used the abbreviation $\mathcal{U}_x = \mathbb{1}_{0:x-2}^{0:L+1} \otimes \mathbb{1}_{x-1}^{0:L,\text{out}} \otimes \mathcal{U} \otimes \mathbb{1}_{x+1}^{\text{in},1:L+1} \otimes \mathbb{1}_{x+2:N}^{0:L+1}$. By using A.1.10, we get

$$\begin{aligned}\rho_x^{\text{mem}} = \mathrm{tr}_{x-1:x}^{0:L,\text{out}}\Big(\mathcal{U}_x\Big(\rho_x^{\text{in}} \otimes \Big(\mathcal{U}_{x-1}\Big(\rho_{x-1}^{\text{in}} \otimes \rho_{x-2}^{\text{mem}} \otimes |0\ldots0\rangle_{x-1}^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_{x-1}^{\dagger}\Big)\Big) \\ \otimes |0\ldots0\rangle_x^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_x^{\dagger}\Big).\end{aligned}$$

Since $\mathcal{U}_{x-1}$ commutes with ρ_x^{in} and $|0\ldots0\rangle_x^{1:L+1}\langle 0\ldots0|$, we are left with

$$\rho_x^{\text{mem}} = \mathrm{tr}_{x-1:x}^{0:L,\text{out}}\Big(\mathcal{U}_x\mathcal{U}_{x-1}\Big(\rho_{x-2}^{\text{mem}} \otimes \rho_{x-1}^{\text{in}} \otimes \rho_x^{\text{in}} \otimes |0\ldots0\rangle_{x-1,x}^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_{x-1}^{\dagger}\mathcal{U}_x^{\dagger}\Big).$$

Repeating this process, we get

$$\rho_x^{\text{mem}} = \mathrm{tr}_{1:x}^{0:L,\text{out}}\Big(\mathcal{U}_{1:x}\Big(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x}\rho_y^{\text{in}} \otimes |0\ldots0\rangle_{1:x}^{1:L+1}\langle 0\ldots0|\Big)\mathcal{U}_{1:x}^{\dagger}\Big), \tag{A.3.10}$$

where again $\mathcal{U}_{x_1:x_2} = \mathcal{U}_{x_2} \dots \mathcal{U}_{x_1}$ for $x_1 \leq x_2$, and $\mathcal{U}_{x_1:x_2} = \mathbb{1}$ for $x_1 > x_2$, and hence

$$\tilde{\rho}_x^{\text{out}} = \text{tr}_x^{0:L,\text{mem}}\left(\text{tr}_{1:x-1}^{0:L,\text{out}}\left(\mathcal{U}_{1:x}\left(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}^\dagger\right)\right).$$

We will now show that actually $\rho_x^{\text{out}} = \tilde{\rho}_x^{\text{out}}$. To do so, look at

$$\begin{aligned}\rho_x^{\text{out}} &= \bar{\text{tr}}_x^{\text{out}}\left(\rho^{\text{OUT}}\right)\\ &= \bar{\text{tr}}_x^{\text{out}}\left(\mathcal{U}_{1:N}\left(\rho^{\text{IN}} \otimes |0\dots0\rangle_{1:N}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:N}{}^\dagger\right).\end{aligned}$$

As $\mathcal{U}_{x+1:N}$ act as the identity on the not-traced-out-layer, we can rotate them around and get

$$\begin{aligned}\rho_x^{\text{out}} &= \bar{\text{tr}}_x^{\text{out}}\left(\mathcal{U}_{1:x}\left(\rho^{\text{IN}} \otimes |0\dots0\rangle_{1:N}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}{}^\dagger\right)\\
&= \bar{\text{tr}}_x^{\text{out}}\left(\mathcal{U}_{1:x}\left(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{N} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:N}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}{}^\dagger\right)\\
&= \text{tr}_{x+2:N}^{0:L+1}\left(\text{tr}_{x+1}^{\text{in},1:L+1}\left(\text{tr}_x^{0:L,\text{mem}}\left(\text{tr}_{1:x-1}^{0:L,\text{out}}\left(\mathcal{U}_{1:x}\left(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}}\right.\right.\right.\right.\right.\\
&\qquad \left.\left.\left.\left.\left.\otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}{}^\dagger \otimes \bigotimes_{y=x+1}^{N} \otimes |0\dots0\rangle_{x+1:N}^{1:L+1}\langle 0\dots0|\right)\right)\right)\right)\\
&= \text{tr}_x^{0:L,\text{mem}}\left(\text{tr}_{1:x-1}^{0:L,\text{out}}\left(\mathcal{U}_{1:x}\left(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}{}^\dagger\right)\right)\\
&\quad \cdot \text{tr}_{x+2:N}^{0:L+1}\left(\text{tr}_{x+1}^{\text{in},1:L+1}\left(\bigotimes_{y=x+1}^{N} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{x+1:N}^{1:L+1}\langle 0\dots0|\right)\right)\\
&= \text{tr}_x^{0:L,\text{mem}}\left(\text{tr}_{1:x-1}^{0:L,\text{out}}\left(\mathcal{U}_{1:x}\left(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\right)\mathcal{U}_{1:x}{}^\dagger\right)\right)\\
&\quad \cdot \text{tr}\left(\bigotimes_{y=x+1}^{N} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{x+1:N}^{1:L+1}\langle 0\dots0|\right)\\
&= \tilde{\rho}_x^{\text{out}} \cdot 1 = \tilde{\rho}_x^{\text{out}},\end{aligned}$$

hence our feed-forward method gives us the correct state, and we can write

$$\rho_x^{\text{out}} = \text{tr}^{\text{mem}}\left(\rho_x^{L+1}\right). \tag{A.3.11}$$

A.3.3 Feed-backward

We define the feed-backward process by

$$\sigma_{xx}^{L+1} = \sigma_x^{\text{out}} \otimes \mathbb{1}_x^{\text{mem}} \tag{A.3.12}$$

$$\sigma_{zx}^{L+1} = \sigma_{z+1\,x}^{\text{mem}} \otimes \mathbb{1}_x^{\text{out}}, \quad z = x-1, ..., 1 \tag{A.3.13}$$

$$\sigma_{zx}^{l} = \mathcal{E}^{l+1\,\dagger}\left(\sigma_{zx}^{l+1}\right), \quad z = x, ..., 1, \quad l = L, ..., 0 \tag{A.3.14}$$

$$\sigma_{zx}^{\text{mem}} = \text{tr}_{z+1}^{\text{in}}\left(\left(\rho_{z+1}^{\text{in}} \otimes \mathbb{1}_z^{\text{mem}}\right)\sigma_{z+1x}^{0}\right), \quad z = x-1, ..., 1. \tag{A.3.15}$$

Like before for the ff DQNN (see Equation (A.1.14)), we know

$$\sigma_{zx}^{l} = \mathrm{tr}_{z}^{l+1:L+1}\Big(\Big(\mathbb{1}_{z}^{l} \otimes |0\dots0\rangle_{z}^{l+1:L+1} \langle 0\dots0| \Big) U_{z}^{l+1:L+1^{\dagger}} \left(\mathbb{1}_{z}^{l:L} \otimes \sigma_{zx}^{L+1} \right) U_{z}^{l+1:L+1} \Big)$$

for $z = x, ..., 1$, which leads to

$$\sigma_{xx}^{l} = \mathrm{tr}_{x}^{l+1:L+1}\Big(\Big(\mathbb{1}_{x}^{l} \otimes |0\dots0\rangle_{x}^{l+1:L+1} \langle 0\dots0| \Big) U_{x}^{l+1:L+1^{\dagger}} \left(\mathbb{1}_{x}^{l:L} \otimes \sigma_{x}^{\mathrm{out}} \otimes \mathbb{1}_{x}^{\mathrm{mem}} \right) U_{x}^{l+1:L+1} \Big) \tag{A.3.16}$$

$$\sigma_{zx}^{l} = \mathrm{tr}_{z}^{l+1:L+1}\Big(\Big(\mathbb{1}_{z}^{l} \otimes |0\dots0\rangle_{z}^{l+1:L+1} \langle 0\dots0| \Big) U_{z}^{l+1:L+1^{\dagger}} \left(\mathbb{1}_{z}^{l:L} \otimes \sigma_{z\,x}^{\mathrm{mem}} \otimes \mathbb{1}_{z}^{\mathrm{out}} \right) U_{z}^{l+1:L+1} \Big) \tag{A.3.17}$$

for $z = x - 1, ..., 1$. Hence, for $z = x - 2, ..., 1$, we get with Equation (A.3.15)

$$\begin{aligned}
\sigma_{zx}^{\mathrm{mem}} &= \mathrm{tr}_{z+1}^{\mathrm{in}}\left(\left(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \right) \sigma_{z+1x}^{0} \right) \\
&= \mathrm{tr}_{z+1}^{\mathrm{in}}\Big(\left(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \right) \mathrm{tr}_{z+1}^{1:L+1}\Big(\Big(\mathbb{1}_{z+1}^{0} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \Big) U_{z+1}^{1:L+1^{\dagger}} \\
&\quad \left(\mathbb{1}_{z+1}^{0:L} \otimes \sigma_{z+1\,x}^{\mathrm{mem}} \otimes \mathbb{1}_{z+1}^{\mathrm{out}} \right) U_{z+1}^{1:L+1} \Big) \Big) \\
&= \mathrm{tr}_{z+1}^{1:L+1,\mathrm{in}}\Big(\left(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes \mathbb{1}_{z+1}^{1:L+1} \right) \Big(\mathbb{1}_{z+1}^{0} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+1}{}^{\dagger} \\
&\quad \left(\mathbb{1}_{z+1}^{0:L} \otimes \sigma_{z+1\,x}^{\mathrm{mem}} \otimes \mathbb{1}_{z+1}^{\mathrm{out}} \right) \mathcal{U}_{z+1} \Big) \\
&= \mathrm{tr}_{z+1}^{1:L+1,\mathrm{in}}\Big(\left(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \right) \mathcal{U}_{z+1}{}^{\dagger} \\
&\quad \left(\mathbb{1}_{z+1}^{0:L} \otimes \sigma_{z+1\,x}^{\mathrm{mem}} \otimes \mathbb{1}_{z+1}^{\mathrm{out}} \right) \mathcal{U}_{z+1} \Big).
\end{aligned}$$

By inserting this in itself, we get for $z = x - 3, ..., 1$

$$\begin{aligned}
\sigma_{zx}^{\mathrm{mem}} &= \mathrm{tr}_{z+1}^{1:L+1,\mathrm{in}}\Big(\Big(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+1}^{\dagger} \Big(\mathbb{1}_{z}^{0:L,\mathrm{out}} \otimes \mathrm{tr}_{z+2}^{1:L+1,\mathrm{in}}\Big(\Big(\rho_{z+2}^{\mathrm{in}} \\
&\quad \otimes \mathbb{1}_{z+1}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+2}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+2}^{\dagger} \Big(\mathbb{1}_{z+2}^{0:L,\mathrm{out}} \otimes \sigma_{z+2\,x}^{\mathrm{mem}} \Big) \mathcal{U}_{z+2} \Big) \Big) \mathcal{U}_{z+1} \Big) \\
&= \mathrm{tr}_{z+1:z+2}^{1:L+1,\mathrm{in}}\Big(\Big(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \otimes \mathbb{1}_{z+2}^{1:L+1,\mathrm{in}} \Big) \mathcal{U}_{z+1}^{\dagger} \Big(\mathbb{1}_{z+1}^{0:L,\mathrm{out}} \\
&\quad \otimes \Big(\Big(\rho_{z+2}^{\mathrm{in}} \otimes \mathbb{1}_{z+1}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+2}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+2}^{\dagger} \Big(\mathbb{1}_{z+2}^{0:L,\mathrm{out}} \otimes \sigma_{z+2\,x}^{\mathrm{mem}} \Big) \mathcal{U}_{z+2} \Big) \Big) \mathcal{U}_{z+1} \Big) \\
&= \mathrm{tr}_{z+1:z+2}^{1:L+1,\mathrm{in}}\Big(\Big(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \otimes \mathbb{1}_{z+2}^{1:L+1,\mathrm{in}} \Big) \mathcal{U}_{z+1}^{\dagger} \Big(\mathbb{1}_{z+1}^{0:L+1} \otimes \rho_{z+2}^{\mathrm{in}} \\
&\quad \otimes |0\dots0\rangle_{z+2}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+2}^{\dagger} \Big(\mathbb{1}_{z+2:z+3}^{0:L,\mathrm{out}} \otimes \sigma_{z+2\,x}^{\mathrm{mem}} \Big) \mathcal{U}_{z+2} \mathcal{U}_{z+1} \Big) \\
&= \mathrm{tr}_{z+1:z+2}^{1:L+1,\mathrm{in}}\Big(\Big(\rho_{z+1}^{\mathrm{in}} \otimes \mathbb{1}_{z}^{\mathrm{mem}} \otimes |0\dots0\rangle_{z+1}^{1:L+1} \langle 0\dots0| \otimes \mathbb{1}_{z+2}^{1:L+1,\mathrm{in}} \Big) \Big(\mathbb{1}_{z+1}^{0:L+1} \otimes \rho_{z+2}^{\mathrm{in}} \\
&\quad \otimes |0\dots0\rangle_{z+2}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+1}^{\dagger} \mathcal{U}_{z+2}^{\dagger} \Big(\mathbb{1}_{z+2:z+3}^{0:L,\mathrm{out}} \otimes \sigma_{z+2\,x}^{\mathrm{mem}} \Big) \mathcal{U}_{z+1:z+2} \Big) \\
&= \mathrm{tr}_{z+1:z+2}^{1:L+1,\mathrm{in}}\Big(\Big(\mathbb{1}_{z}^{\mathrm{mem}} \otimes \bigotimes_{y=z+1}^{z+2} \rho_{y}^{\mathrm{in}} \otimes |0\dots0\rangle_{z+1:z+2}^{1:L+1} \langle 0\dots0| \Big) \mathcal{U}_{z+1:z+2}^{\dagger} \\
&\quad \Big(\mathbb{1}_{z+2:z+3}^{0:L,\mathrm{out}} \otimes \sigma_{z+2\,x}^{\mathrm{mem}} \Big) \mathcal{U}_{z+1:z+2} \Big).
\end{aligned}$$

Doing this recursively, we are left with

$$\sigma_{zx}^{\text{mem}} = \text{tr}_{z+1:x-1}^{1:L+1,\text{in}}\Big(\Big(\mathbb{1}_z^{\text{mem}} \otimes \bigotimes_{y=z+1}^{x-1} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{z+1:x-1}^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_{z+1:x-1}^\dagger \Big(\mathbb{1}_{z+2:x-1}^{0:L,\text{out}} \otimes \sigma_{x-1\,x}^{\text{mem}}\Big)\mathcal{U}_{z+1:x-1}\Big). \tag{A.3.18}$$

For $\sigma_{x-1\,x}^{\text{mem}}$, we do the same first steps as for σ_{zx}^{mem}, only that we have $\sigma_{xx}^{L+1} = \sigma_x^{\text{out}} \otimes \mathbb{1}_x^{\text{mem}}$ instead of $\sigma_{zx}^{L+1} = \sigma_{z+1\,x}^{\text{mem}} \otimes \mathbb{1}^{\text{out}}$, which leads to

$$\begin{aligned}\sigma_{x-1\,x}^{\text{mem}} &= \text{tr}_x^{\text{in}}\Big(\Big(\rho_x^{\text{in}} \otimes \mathbb{1}_{x-1}^{\text{mem}}\Big)\text{tr}_x^{1:L+1}\Big(\Big(\mathbb{1}_x^0 \otimes |0\dots0\rangle_x^{1:L+1} \langle 0\dots0| \Big)U_x^{1:L+1\,\dagger} \\ &\quad \Big(\mathbb{1}_x^{0:L} \otimes \sigma_x^{\text{out}} \otimes \mathbb{1}_x^{\text{mem}}\Big)U_x^{1:L+1}\Big)\Big) \\ &= \text{tr}_x^{1:L+1,\text{in}}\Big(\Big(\rho_x^{\text{in}} \otimes \mathbb{1}_{x-1}^{\text{mem}} \otimes |0\dots0\rangle_x^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_x^\dagger\Big(\mathbb{1}_x^{0:L,\text{mem}} \otimes \sigma_x^{\text{out}}\Big)\mathcal{U}_x\Big).\end{aligned}$$

Together with (A.3.18), we have

$$\begin{aligned}\sigma_{zx}^{\text{mem}} = \text{tr}_{z+1:x-1}^{1:L+1,\text{in}}\Big(\Big(\mathbb{1}_z^{\text{mem}} \otimes \bigotimes_{y=z+1}^{x-1} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{z+1:x-1}^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_{z+1:x-1}^\dagger\Big(\mathbb{1}_{z+2:x-1}^{0:L,\text{out}} \\ \otimes \text{tr}_x^{1:L+1,\text{in}}\Big(\Big(\rho_x^{\text{in}} \otimes \mathbb{1}_{x-1}^{\text{mem}} \otimes |0\dots0\rangle_x^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_x^\dagger\Big(\mathbb{1}_x^{0:L,\text{mem}} \otimes \sigma_x^{\text{out}}\Big) \\ \mathcal{U}_x\Big)\Big)\mathcal{U}_{z+1:x-1}\Big).\end{aligned}$$

Just like before, this is

$$\sigma_{zx}^{\text{mem}} = \text{tr}_{z+1:x}^{1:L+1,\text{in}}\Big(\Big(\mathbb{1}_z^{\text{mem}} \otimes \bigotimes_{y=z+1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{z+1:x}^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_{z+1:x}^\dagger \Big(\mathbb{1}_{z+2:x-1}^{0:L,\text{out}} \otimes \mathbb{1}_x^{0:L,\text{mem}} \otimes \sigma_x^{\text{out}}\Big)\mathcal{U}_{z+1:x}\Big) \tag{A.3.19}$$

for $z = x-1, ..., 1$. Inserting this into (A.3.17), we get

$$\begin{aligned}\sigma_{zx}^l = \text{tr}_z^{l+1:L+1}\Big(\Big(\mathbb{1}_z^l \otimes |0\dots0\rangle_z^{l+1:L+1} \langle 0\dots0|\Big)U_z^{l+1:L+1\,\dagger}\Big(\mathbb{1}_z^{l:L} \otimes \text{tr}_{z+1:x}^{1:L+1,\text{in}}\Big(\Big(\mathbb{1}_z^{\text{mem}} \\ \otimes \bigotimes_{y=z+1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{z+1:x}^{1:L+1} \langle 0\dots0| \Big)\mathcal{U}_{z+1:x}^\dagger\Big(\mathbb{1}_x^{0:L,\text{mem}} \otimes \sigma_x^{\text{out}}\Big)\mathcal{U}_{z+1:x}\Big) \\ \otimes \mathbb{1}_z^{\text{out}}\Big)U_z^{l+1:L+1}\Big).\end{aligned}$$

Like before, it is

$$\begin{aligned}\sigma_{zx}^l = \text{tr}_z^{l+1:L+1}\Big(\text{tr}_{z+1:x}^{1:L+1,\text{in}}\Big(\Big(\mathbb{1}_z^l \otimes |0\dots0\rangle_z^{l+1:L+1} \langle 0\dots0| \otimes \bigotimes_{y=z+1}^{x} \rho_y^{\text{in}} \\ \otimes |0\dots0\rangle_{z+1:x}^{1:L+1} \langle 0\dots0| \Big)U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:x}^\dagger\Big(\mathbb{1}_{z+2:x-1}^{0:L,\text{out}} \otimes \mathbb{1}_x^{0:L,\text{mem}} \otimes \sigma_x^{\text{out}}\Big) \\ \mathcal{U}_{z+1:x}U_z^{l+1:L+1}\Big)\Big).\end{aligned} \tag{A.3.20}$$

A.3.4 Change of the Cost function and Proof of Proposition 4.1 for Pure Target Outputs and One Run

In each training step, we again update our unitaries U_j^l according to

$$U_j^l \mapsto U_j^{l\,\prime} = e^{i\epsilon K_j^l} U_j^l \tag{A.3.21}$$

for some small number ϵ and a hermitian matrix K_j^l. First, let us calculate $\delta\rho_x^{\text{out}}$ like before in Section A.1.3. It is

$$\begin{aligned}
\rho_x^{\text{out}\,\prime} &= \text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{1:x-1}^{0:L,\text{out}}\Big(\mathcal{U}'_{1:x}\Big(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\Big){\mathcal{U}'_{1:x}}^{\dagger}\Big)\Big) \\
&= \rho_x^{\text{out}} + i\epsilon \sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l} \text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{1:x-1}^{0:L,\text{out}}\Big(\mathcal{U}_{z+1:x} U_z^{l+1:L+1} U_{j+1:m_l\,z}^l \Big[K_{j\,z}^l, U_{1:j\,z}^l U_z^{1:l-1} \\
&\quad \mathcal{U}_{1:z-1}\Big(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\Big)\mathcal{U}_{1:z-1}^{\dagger} {U_z^{1:l-1}}^{\dagger} {U_{1:j\,z}^l}^{\dagger}\Big] \\
&\quad {U_{j+1:m_l\,z}^l}^{\dagger} {U_z^{l+1:L+1}}^{\dagger} \mathcal{U}_{z+1:x}^{\dagger}\Big)\Big) + \mathcal{O}(\epsilon^2) \\
&= \rho_x^{\text{out}} + \epsilon\delta\rho_x^{\text{out}} + \mathcal{O}(\epsilon^2)
\end{aligned}$$

with

$$\begin{aligned}
\delta\rho_x^{\text{out}} = i \sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l} \text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{1:x-1}^{0:L,\text{out}}\Big(\mathcal{U}_{z+1:x} U_z^{l+1:L+1} U_{j+1:m_l\,z}^l \Big[K_{j\,z}^l, U_{1:j\,z}^l U_z^{1:l-1} \\
\mathcal{U}_{1:z-1}\Big(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:x}^{1:L+1}\langle 0\dots0|\Big)\mathcal{U}_{1:z-1}^{\dagger} {U_z^{1:l-1}}^{\dagger} {U_{1:j\,z}^l}^{\dagger}\Big] \\
{U_{j+1:m_l\,z}^l}^{\dagger} {U_z^{l+1:L+1}}^{\dagger} \mathcal{U}_{z+1:x}^{\dagger}\Big)\Big).
\end{aligned}$$

Like before in Section A.1.3 (see e.g. Equation (A.1.23)), we can pull part of the traces in and get

$$\begin{aligned}
\delta\rho_x^{\text{out}} = i \sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l} \text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{z:x-1}^{0:L,\text{out}}\Big(\mathcal{U}_{z+1:x} U_z^{l+1:L+1} U_{j+1:m_l\,z}^l \Big[K_{j\,z}^l, U_{1:j\,z}^l U_z^{1:l-1} \\
\Big(\bigotimes_{y=z}^{x} \rho_y^{\text{in}} \otimes \text{tr}_{1:z-1}^{0:L,\text{out}}\Big(\mathcal{U}_{1:z-1}\Big(\rho_0^{\text{mem}} \otimes \bigotimes_{y=1}^{z-1} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{1:z-1}^{1:L+1}\langle 0\dots0|\Big)\mathcal{U}_{1:z-1}^{\dagger}\Big) \\
\otimes |0\dots0\rangle_{z:x}^{1:L+1}\langle 0\dots0|\Big) {U_z^{1:l-1}}^{\dagger} {U_{1:j\,z}^l}^{\dagger}\Big] {U_{j+1:m_l\,z}^l}^{\dagger} {U_z^{l+1:L+1}}^{\dagger} \mathcal{U}_{z+1:x}^{\dagger}\Big)\Big).
\end{aligned}$$

Using A.3.10 leads to

$$\begin{aligned}
\delta\rho_x^{\text{out}} = i \sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l} \text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{z:x-1}^{0:L,\text{out}}\Big(\mathcal{U}_{z+1:x} U_z^{l+1:L+1} U_{j+1:m_l\,z}^l \Big[K_{j\,z}^l, \\
U_{1:j\,z}^l U_z^{1:l-1}\Big(\rho_{z-1}^{\text{mem}} \otimes \bigotimes_{y=z}^{x} \rho_y^{\text{in}} \otimes |0\dots0\rangle_{z:x}^{1:L+1}\langle 0\dots0|\Big) {U_z^{1:l-1}}^{\dagger} {U_{1:j\,z}^l}^{\dagger}\Big] \\
{U_{j+1:m_l\,z}^l}^{\dagger} {U_z^{l+1:L+1}}^{\dagger} \mathcal{U}_{z+1:x}^{\dagger}\Big)\Big).
\end{aligned}$$

Pulling in traces and using Equation (A.3.9), we are left with

$$\begin{aligned}
\delta\rho_x^{\text{out}} = i\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{z+1:x-1}^{0:L,\text{out}}\Big(\text{tr}_z^{l-1:L,\text{out}}\Big(&\mathcal{U}_{z+1:x}U_z^{l+1:L+1}U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},\\
&U_{1:j\,z}^{l}\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\text{in}}\otimes\text{tr}_z^{0:l-2}\Big(U_z^{1:l-1}\Big(\rho_{z-1}^{\text{mem}}\otimes\rho_z^{\text{in}}\otimes|0\dots0\rangle_z^{1:l-1}\langle0\dots0|\Big)U_z^{1:l-1\dagger}\Big)\\
&\otimes|0\dots0\rangle_z^{l:L+1}\langle0\dots0|\otimes|0\dots0\rangle_{z+1:x}^{1:L+1}\langle0\dots0|\Big)U_{1:j\,z}^{l\,\dagger}\Big]U_{j+1:m_l\,z}^{l\,\dagger}U_z^{l+1:L+1\dagger}\\
&\mathcal{U}_{z+1:x}^{\dagger}\Big)\Big)\Big)\\
= i\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{z+1:x-1}^{0:L,\text{out}}\Big(\text{tr}_z^{l-1:L,\text{out}}\Big(&\mathcal{U}_{z+1:x}U_z^{l+1:L+1}U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\\
&\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\text{in}}\otimes\rho_z^{l-1}\otimes|0\dots0\rangle_z^{l:L+1}\langle0\dots0|\otimes|0\dots0\rangle_{z+1:x}^{1:L+1}\langle0\dots0|\Big)U_{1:j\,z}^{l\,\dagger}\Big]\\
&U_{j+1:m_l\,z}^{l\,\dagger}U_z^{l+1:L+1\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big)\Big)\Big).
\end{aligned}$$

It is

$$\begin{aligned}
\delta C &= \lim_{\epsilon\to0}\frac{C'-C}{\epsilon}\\
&= \lim_{\epsilon\to0}\frac{1-\frac{1}{N}\sum_{x=1}^{N}\text{tr}\big(\rho_x^{\text{out}\prime}\sigma_x^{\text{out}}\big)-\big(1-\frac{1}{N}\sum_{x=1}^{N}\text{tr}\big(\rho_x^{\text{out}}\sigma_x^{\text{out}}\big)\big)}{\epsilon}\\
&= -\frac{1}{N}\sum_{x=1}^{N}\lim_{\epsilon\to0}\frac{\text{tr}\left(\left(\delta\rho_x^{\text{out}}\right)\sigma_x^{\text{out}}\right)+\mathcal{O}\left(\epsilon^2\right)}{\epsilon}\\
&= -\frac{i}{N}\sum_{x=1}^{N}\text{tr}\left(\left(\delta\rho_x^{\text{out}}\right)\sigma_x^{\text{out}}\right)\\
&= -\frac{i}{N}\sum_{x=1}^{N}\text{tr}\Big(\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}_x^{0:L,\text{mem}}\Big(\text{tr}_{z+1:x-1}^{0:L,\text{out}}\Big(\text{tr}_z^{l-1:L,\text{out}}\Big(\mathcal{U}_{z+1:x}U_z^{l+1:L+1}U_{j+1:m_l\,z}^{l}\\
&\quad\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\text{in}}\otimes\rho_z^{l-1}\otimes|0\dots0\rangle_z^{l:L+1}\langle0\dots0|\otimes|0\dots0\rangle_{z+1:x}^{1:L+1}\langle0\dots0|\Big)U_{1:j\,z}^{l\,\dagger}\Big]\\
&\quad U_{j+1:m_l\,z}^{l\,\dagger}U_z^{l+1:L+1\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big)\Big)\Big)\sigma_x^{\text{out}}\Big)\\
&= -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}\Big(\mathcal{U}_{z+1:x}U_z^{l+1:L+1}U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},\\
&\quad U_{1:j\,z}^{l}\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\text{in}}\otimes\rho_z^{l-1}\otimes|0\dots0\rangle_z^{l:L+1}\langle0\dots0|\otimes|0\dots0\rangle_{z+1:x}^{1:L+1}\langle0\dots0|\Big)U_{1:j\,z}^{l\,\dagger}\Big]\\
&\quad U_{j+1:m_l\,z}^{l\,\dagger}U_z^{l+1:L+1\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big(\mathbb{1}_x^{0:L,\text{mem}}\otimes\mathbb{1}_{z+1:x-1}^{0:L,\text{out}}\otimes\mathbb{1}_z^{l-1:L,\text{out}}\otimes\sigma_x^{\text{out}}\Big)\Big).
\end{aligned}$$

By using the cyclic rule of trace, we get

$$\begin{aligned}
\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big(\mathbb{1}_x^{0:L,\mathrm{mem}}\otimes\mathbb{1}_{z+1:x-1}^{0:L,\mathrm{out}}\otimes\mathbb{1}_z^{l-1:L,\mathrm{out}}\otimes\sigma_x^{\mathrm{out}}\Big)\\
&\mathcal{U}_{z+1:x}U_z^{l+1:L+1}U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\mathrm{in}}\otimes\rho_z^{l-1}\otimes|0\ldots0\rangle_z^{l:L+1}\langle 0\ldots0|\\
&\otimes|0\ldots0\rangle_{z+1:x}^{1:L+1}\langle 0\ldots0|\Big)U_{1:j\,z}^{l}{}^{\dagger}\Big]U_{j+1:m_l\,z}^{l}{}^{\dagger}\Big)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big(\mathbb{1}_x^{0:L,\mathrm{mem}}\otimes\mathbb{1}_{z+1:x-1}^{0:L,\mathrm{out}}\otimes\mathbb{1}_z^{l-1:L,\mathrm{out}}\otimes\sigma_x^{\mathrm{out}}\Big)\\
&\mathcal{U}_{z+1:x}U_z^{l+1:L+1}\Big(U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\Big(\rho_z^{l-1}\otimes|0\ldots0\rangle_z^{l}\langle 0\ldots0|\Big)U_{1:j\,z}^{l}{}^{\dagger}\Big]U_{j+1:m_l\,z}^{l}{}^{\dagger}\\
&\otimes\mathbb{1}_z^{l+1:L+1}\otimes\mathbb{1}_{z+1:x}^{1:L+1,\mathrm{in}}\Big)\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\mathrm{in}}\otimes\mathbb{1}_z^{l-1:l}\otimes|0\ldots0\rangle_z^{l+1:L+1}\langle 0\ldots0|\\
&\otimes|0\ldots0\rangle_{z+1:x}^{1:L+1}\langle 0\ldots0|\Big)\Big).
\end{aligned}$$

This is

$$\begin{aligned}
\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\mathrm{in}}\otimes\mathbb{1}_z^{l-1:l}\otimes|0\ldots0\rangle_z^{l+1:L+1}\langle 0\ldots0|\\
&\otimes|0\ldots0\rangle_{z+1:x}^{1:L+1}\langle 0\ldots0|\Big)U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:x}^{\dagger}\Big(\mathbb{1}_x^{0:L,\mathrm{mem}}\otimes\mathbb{1}_{z+1:x-1}^{0:L,\mathrm{out}}\otimes\mathbb{1}_z^{l-1:L,\mathrm{out}}\otimes\sigma_x^{\mathrm{out}}\Big)\\
&\mathcal{U}_{z+1:x}U_z^{l+1:L+1}\Big(U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\Big(\rho_z^{l-1}\otimes|0\ldots0\rangle_z^{l}\langle 0\ldots0|\Big)U_{1:j\,z}^{l}{}^{\dagger}\Big]U_{j+1:m_l\,z}^{l}{}^{\dagger}\\
&\otimes\mathbb{1}_z^{l+1:L+1}\otimes\mathbb{1}_{z+1:x}^{1:L+1,\mathrm{in}}\Big)\Big)\\
= & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\Big(\mathbb{1}_z^{l-1}\otimes\operatorname{tr}_z^{l+1:L+1}\Big(\operatorname{tr}_{z+1:x}^{1:L+1,\mathrm{in}}\Big(\Big(\bigotimes_{y=z+1}^{x}\rho_y^{\mathrm{in}}\otimes\mathbb{1}_z^{l}\\
&\otimes|0\ldots0\rangle_z^{l+1:L+1}\langle 0\ldots0|\otimes|0\ldots0\rangle_{z+1:x}^{1:L+1}\langle 0\ldots0|\Big)U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:x}^{\dagger}\\
&\Big(\mathbb{1}_x^{0:L,\mathrm{mem}}\otimes\mathbb{1}_{z+1:x-1}^{0:L,\mathrm{out}}\otimes\mathbb{1}_z^{l-1:L,\mathrm{out}}\otimes\sigma_x^{\mathrm{out}}\Big)\mathcal{U}_{z+1:x}U_z^{l+1:L+1}\Big)\Big)\Big)\\
&U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\Big(\rho_z^{l-1}\otimes|0\ldots0\rangle_z^{l}\langle 0\ldots0|\Big)U_{1:j\,z}^{l}{}^{\dagger}\Big]U_{j+1:m_l\,z}^{l}{}^{\dagger}\Big).
\end{aligned}$$

With (A.3.20), we get

$$\begin{aligned}
\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}\Big(\Big(\mathbb{1}_z^{l-1}\otimes\sigma_{zx}^{l}\Big)U_{j+1:m_l\,z}^{l}\Big[K_{j\,z}^{l},U_{1:j\,z}^{l}\\
&\Big(\rho_z^{l-1}\otimes|0\ldots0\rangle_z^{l}\langle 0\ldots0|\Big)U_{1:j\,z}^{l}{}^{\dagger}\Big]U_{j+1:m_l\,z}^{l}{}^{\dagger}\Big).
\end{aligned}$$

We can drop the subscript z for the unitaries, identities, and ground states, as we are only acting on network z. Then, using $\operatorname{tr}(A[B,C]D)=\operatorname{tr}([C,DA]B)$ for $A,B,C,D\in\mathcal{B}(\mathcal{H})$ (see

Equation (A.1.25)) for some Hilbert space $\mathcal{H}$, we have

$$\begin{aligned}\delta C = & -\frac{i}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\mathrm{tr}\Big(\Big(\mathbb{1}_z^{l-1}\otimes\sigma_{zx}^l\Big)U_{j+1:m_l}^l\Big[K_j^l, U_{1:j}^l \\ & \Big(\rho_z^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|^l\Big)U_{1:j}^{l\,\dagger}\Big]U_{j+1:m_l}^{l\,\dagger}\Big) \\ = & -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\mathrm{tr}\Big(M_j^l K_j^l\Big)\end{aligned}$$

with

$$M_j^l = -\frac{1}{N}\sum_{x=1}^{N}\sum_{z=1}^{x}\Big[U_{1:j}^l\Big(\rho_z^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|^l\Big)U_{1:j}^{l\,\dagger}, U_{j+1:m_l}^{l\,\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_{zx}^l\Big)U_{j+1:m_l}^l\Big] \tag{A.3.22}$$

where we again used Equation (A.1.25). This concludes the proof of Proposition 4.1 for $M = 1$ and pure target output states.

Using Proposition 4.2, we get

$$K_j^l = i\frac{2^{m_{l-1}}i}{\lambda}\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big).$$

Hence, in each step, we perform the update

$$U_j^l \mapsto \exp\left(P_j^l\right)U_j^l$$

with

$$\begin{aligned}P_j^l &= i\epsilon K_j^l \\ &= -2^{m_{l-1}}\eta\,\mathrm{tr}_{1:j-1}^l\Big(\mathrm{tr}_{j+1:m_l}^l\left(M_j^l\right)\Big),\end{aligned}$$

where $\eta = \frac{\epsilon}{\lambda}$ is the learning rate.

A.3.5 The algorithm

To reintroduce α, we just have to average, i.e.,

$$M_j^l = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\sum_{z=1}^{x}\Big[U_{1:j}^l\Big(\rho_{x\,\alpha}^{l-1}\otimes|0\ldots0\rangle^l\langle 0\ldots0|^l\Big)U_{1:j}^{l\,\dagger}, U_{j+1:m_l}^{l\,\dagger}\Big(\mathbb{1}^{l-1}\otimes\sigma_{zx\,\alpha}^l\Big)U_{j+1:m_l}^l\Big]. \tag{A.3.23}$$

This leaves us with algorithm 16 and concludes the proof of Proposition 4.1 for pure target output states.

A.3.6 Mixed Target Outputs

For mixed target outputs, i.e., $\sigma_{x\,\alpha}^{\mathrm{out}} \in \mathcal{D}(\mathcal{H}^{\mathrm{out}})$, we have

$$C_{\text{local, mixed},S} = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\mathrm{tr}((\rho_{x\,\alpha}^{\mathrm{out}} - \sigma_{x\,\alpha}^{\mathrm{out}})^2). \tag{A.3.24}$$

Algorithm 16 Classical Training Algorithm of DQRNNs with pure target and local cost

Input: $((\rho_{0\,\alpha}^{\text{mem}}, (\rho_{x\,\alpha}^{\text{in}}, |\psi_{x\,\alpha}^{\text{out}}\rangle))_{x=1,\dots,N})_{\alpha=1,\dots,M}$, $(m_l)_{l=0,\dots,L+1}, m_{\text{mem}}$ or $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$, η, T

▷ *Training set, network architecture or unitaries, learning rate, number of training steps* ◁

if $(m_l)_{l=1,\dots,L+1}$ not given **then** ▷ *Calculate architecture if none is given*

  for $l = 0, \dots,$Len$(((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1})$ **do**

    $m_l \leftarrow$Len$((U_j^l)_{j=1,\dots,m_l})$

  $L \leftarrow$Len$((m_l)_{l=1,\dots,L+1})-1$

else if $((U_j^l)_{j=1,\dots,m_l})_{l=1,\dots,L+1}$ not given **then** ▷ *Initialise unitaries if none are given*

  for l=1,...,L+1 **do**

    for $j = 1, \dots, m_l$ **do**

      $U_j^l \leftarrow$RandomUnitaryHaar$(m_{l-1}+1$ qudits$)$ ▷ *Random unitary w.r.t. Haar measure on $m_{l-1}+1$ qudits*

      $U_j^l \leftarrow \mathbb{1}_{1:j-1}^l \otimes U_j^l \otimes \mathbb{1}_{j+1:m_l}^l$

for t=1,...,T **do**

  for $\alpha = 1, \dots, M$ **do** ▷ *Feed-forward*

    for x=1,...,N **do**

      $\rho_{x\,\alpha}^0 \leftarrow \rho_{x-1\,\alpha}^{\text{mem}} \otimes \rho_{x\,\alpha}^{\text{in}}$

      for l=1,...,L+1 **do**

        $\rho_{x\,\alpha}^l \leftarrow \text{tr}^{l-1}\left(U^l\left(\rho_{x\,\alpha}^{l-1} \otimes |0\dots0\rangle^l\langle 0\dots0|\right)U^{l\dagger}\right)$

      $\rho_{x\,\alpha}^{\text{mem}} \leftarrow \text{tr}^{\text{out}}\left(\rho_{x\,\alpha}^{L+1}\right)$

      $\rho_{x\,\alpha}^{\text{out}} \leftarrow \text{tr}^{\text{mem}}\left(\rho_{x\,\alpha}^{L+1}\right)$

  for $\alpha = 1, \dots, M$ **do** ▷ *Feed-backward*

    for x=1,...,N **do**

      $\sigma_{xx\,\alpha}^{L+1} \leftarrow |\psi_{x\,\alpha}^{\text{out}}\rangle\langle\psi_{x\,\alpha}^{\text{out}}| \otimes \mathbb{1}^{\text{mem}}$

      for z=x,...,1 **do**

        for l=0,...,L **do**

          $\sigma_{zx\,\alpha}^l \leftarrow \text{tr}^{l+1}\left(\left(\mathbb{1}^l \otimes |0\dots0\rangle^{l+1}\langle 0\dots0|\right)U^{l+1\dagger}\left(\mathbb{1}^l \otimes \sigma_{zx\,\alpha}^{l+1}\right)U^{l+1}\right)$

        $\sigma_{z-1\,x\,\alpha}^{\text{mem}} \leftarrow \text{tr}_z^{1,\text{in}}\left(\left(\rho_{z\,\alpha}^{\text{in}} \otimes \mathbb{1}_{z-1}^{\text{mem}} \otimes |0\dots0\rangle_z^1\langle 0\dots0|\right)U_z^{1\dagger}\left(\mathbb{1}_z^0 \otimes \sigma_{zx\,\alpha}^1\right)U_z^1\right)$

        $\sigma_{z-1\,x\,\alpha}^{L+1} \leftarrow \mathbb{1}^{\text{out}} \otimes \sigma_{z-1\,x\,\alpha}^{\text{mem}}$

  for l=1,...,L+1 **do** ▷ *Update matrices*

    for $j = 1, \dots, m_l$ **do**

      $$M_j^l \leftarrow \frac{1}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\sum_{z=1}^{x}\Big[U_{1:j}^l\left(\rho_{x\,\alpha}^{l-1} \otimes |0\dots0\rangle^l\langle 0\dots0|\right)U_{1:j}^{l\,\dagger}, U_{j+1:m_l}^{l\,\dagger}\left(\mathbb{1}^{l-1} \otimes \sigma_{zx\,\alpha}^l\right)U_{j+1:m_l}^l\Big]$$

      $P_j^l \leftarrow -2^{m_{l-1}}\eta\,\text{tr}_{1:j-1}^l\left(\text{tr}_{j+1:m_l}^l\left(M_j^l\right)\right)$

      $U_j^l \leftarrow \exp\left(P_j^l\right)U_j^l$

  $C_t \leftarrow 1 - \frac{1}{NM}\sum_{\alpha=1}^{M}\sum_{x=1}^{N}\text{tr}\left(\rho_{x\,\alpha}\sigma_{xx\,\alpha}\right)$ ▷ *Calculate Cost*

As for the DQNNs, if we look at mixed outputs, we have

$$\delta C = \frac{2}{M}\sum_{\alpha=1}^{M}\frac{1}{N_\alpha}\sum_{x=1}^{N_\alpha}\mathrm{tr}((\delta\rho_{x\,\alpha}^{\mathrm{out}})(\rho_{x\,\alpha}^{\mathrm{out}}-\sigma_{x\,\alpha}^{\mathrm{out}}), \tag{A.3.25}$$

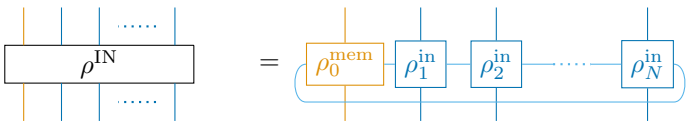
i.e., we just have to replace $\sigma_x^{L+1}=\sigma_{x\,\alpha}^{\mathrm{out}}$ with $\sigma_x^{L+1}=(\sigma_{x\,\alpha}^{\mathrm{out}}-\rho_{x\,\alpha}^{\mathrm{out}})$ in the feed-forward process and get Algorithm 10 and Proposition 4.1 for mixed target output states..

A.4 DQRNN with Global Cost and Pure Target Output

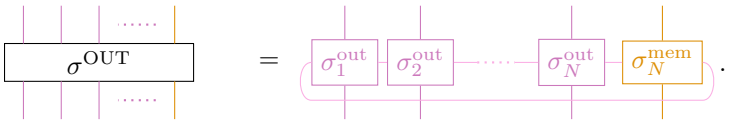
We now drop the assumption that the global inputs and outputs of the DQRNN are product states. Hence, the training set is generally of the form

$$S=\{S_\alpha\}_{\alpha=1}^{M}=\left\{\left(\rho_\alpha^{\mathrm{IN}},\sigma_\alpha^{\mathrm{OUT}}\right)\right\}_{\alpha=1}^{M}=\in\bigtimes_{\alpha=1}^{M}\left(\mathcal{D}(\mathcal{H}_{N_\alpha}^{\mathrm{IN}})\times\mathcal{D}(\mathcal{H}_{N_\alpha}^{\mathrm{OUT}})\right),$$

where we assume our outputs to be pure and write both inputs and outputs as MPOs, i.e.,



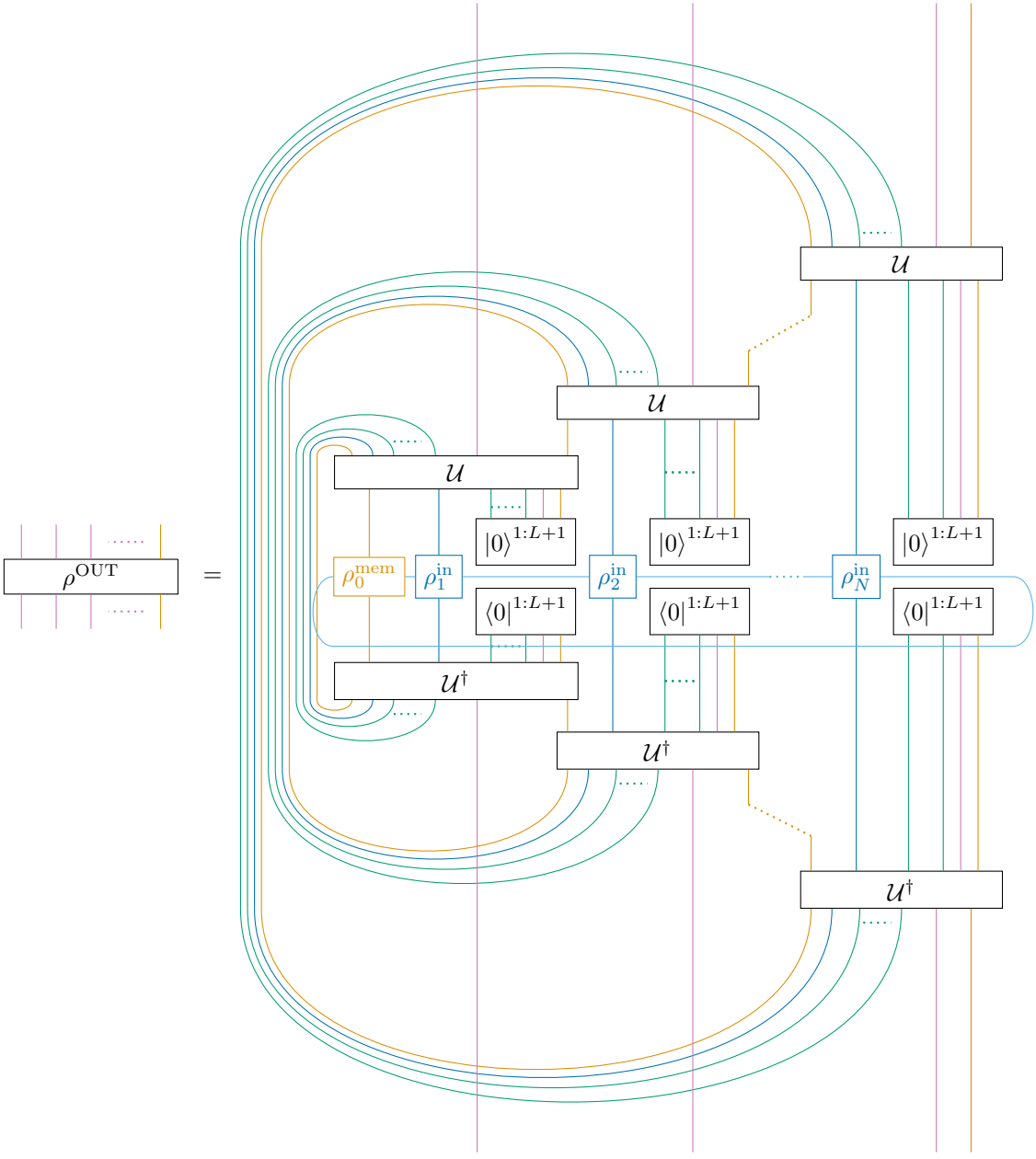
(A.4.1)



(A.4.2)

In order to keep the derivations shorter when deriving the training algorithm, we again set $M=1$ until mentioned otherwise. We use different colors for the input, memory, output, and hidden layers here and in the following. If the last memory output is not given, choose

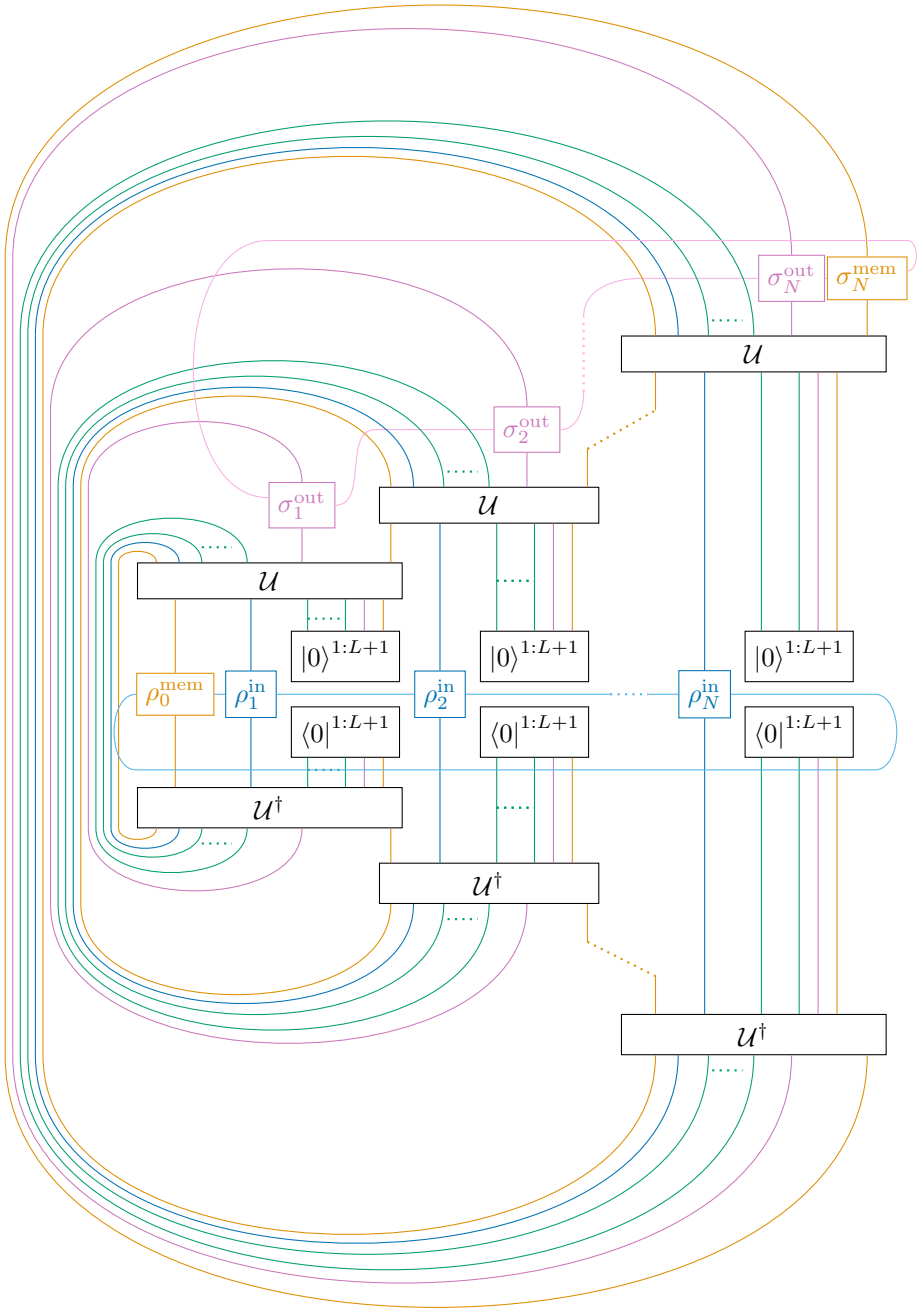
$\sigma_N^{\text{mem}} = \mathbb{1}$. The output of the network is then given by



(A.4.3)

The cost is given by $C_{\text{global, pure},S} = 1 - \text{tr}\left(\sigma_\alpha^{\text{OUT}} \rho_\alpha^{\text{OUT}}\right)$. In TNN, we can write

$C \;=\; 1 -$

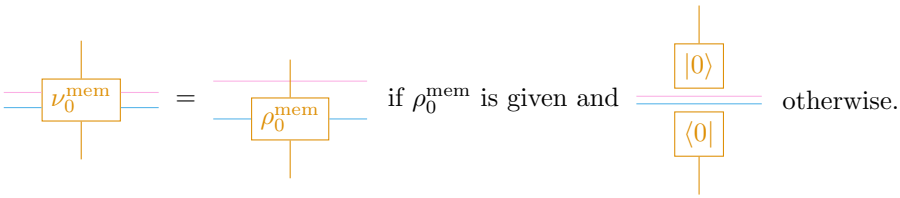


(A.4.4)

A.4.1 Feed-forward

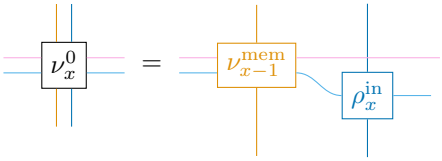
We again define the feed-forward procedure similar to before, but now in TNN.

First, we set up the first memory state by setting



(A.4.5)

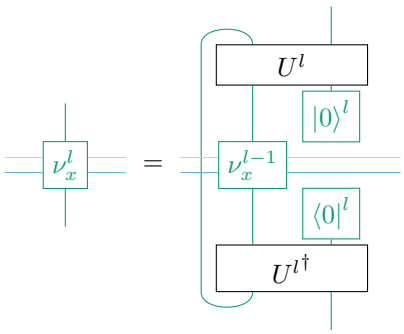
In order to get the total input of each DQNN iteration, we set



(A.4.6)

for $x = 1, ..., N$.

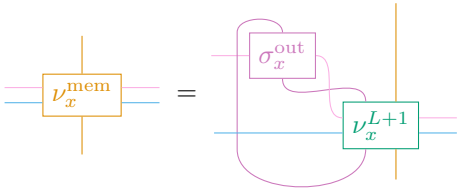
This total input then has to be propagated through the DQNN using $\mathcal{E}^l$ as in Equation (A.1.7). Hence, we set



(A.4.7)

for $l = 1, ..., L + 1$.

To get the cost or derivatives of the cost, we already contract the output legs of ν_x^{L+1} with σ_x^{out} and trace out the physical output legs. The remaining memory expression then is

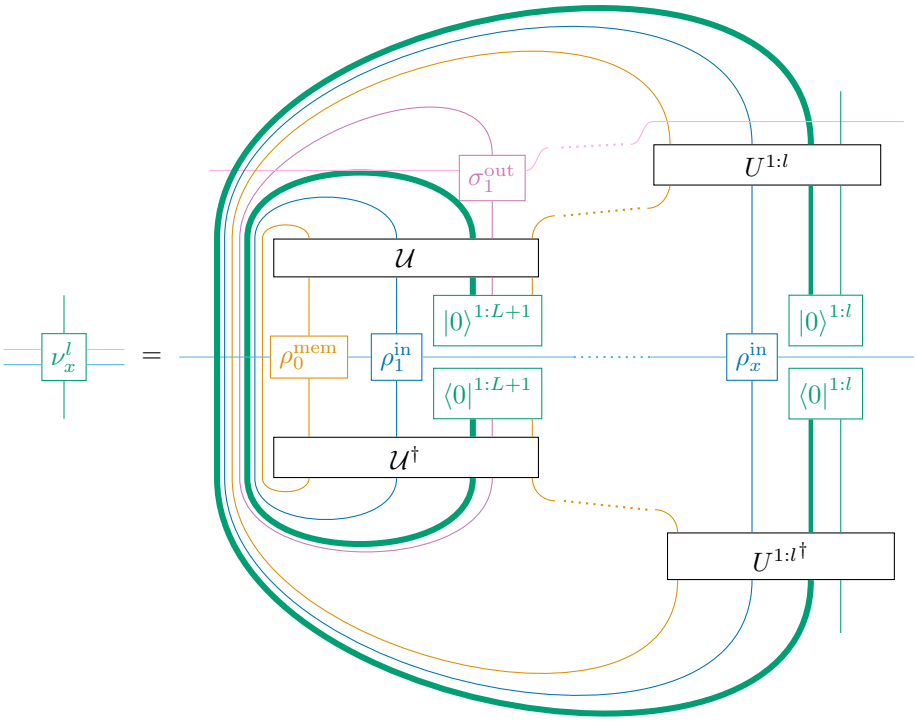


(A.4.8)

which leaves us with open memory and virtual input and output legs.

In the following, we again use $\mathcal{U}_{x:y} = \mathcal{U}_y \dots \mathcal{U}_x$, $U^{l_1:l_2} = U^{l_2} \dots U^{l_1}$, $U^l_{j_1:j_2} = U^l_{j_2} \dots U^l_{j_1}$ for $x > y, l_2 > l_1, j_2 > j_1$ in short-hand.

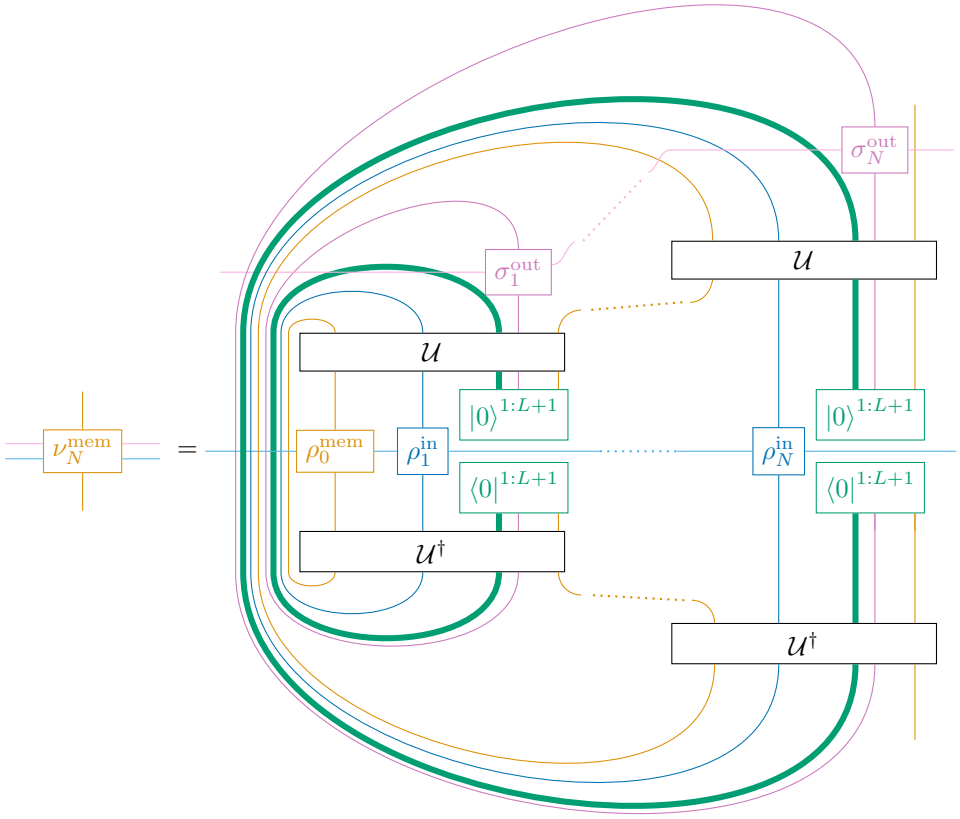
Iteratively inserting the above equations into each other, starting from ν^l_x in Equation (A.4.7) and ending with the definition of ν^{mem}_0 in Equation (A.4.5), leaves us with



(A.4.9)

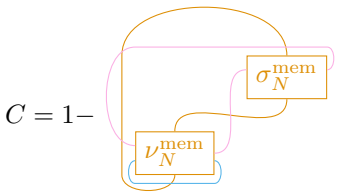
for $x = 1, ..., N$ and $l = 1, ...L + 1$.

By going up to $x = N$ and $l = L + 1$, and then applying Equation (A.4.8), this leads to



(A.4.10)

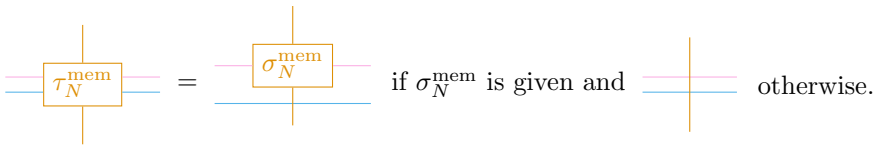
which again leads to



(A.4.11)

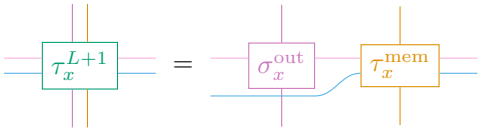
A.4.2 Feed-backward

First, we, similarly to before, set up the last memory state by setting



(A.4.12)

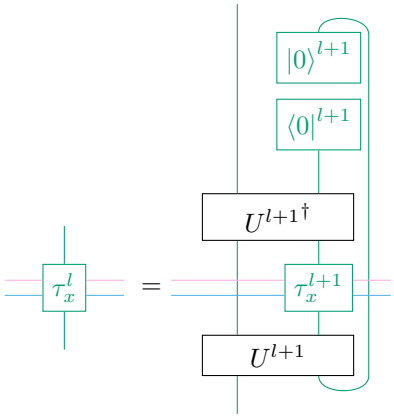
In order to get the total input of each DQNN iteration, we set



(A.4.13)

for $x = N, ..., 1$.

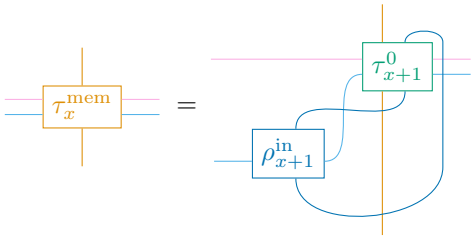
This total output then has to be propagated back through the DQNN using $\mathcal{E}^{l\dagger}$ as in Equation (A.1.9). Hence, we set



(A.4.14)

for $l = L + 1, ..., 0$.

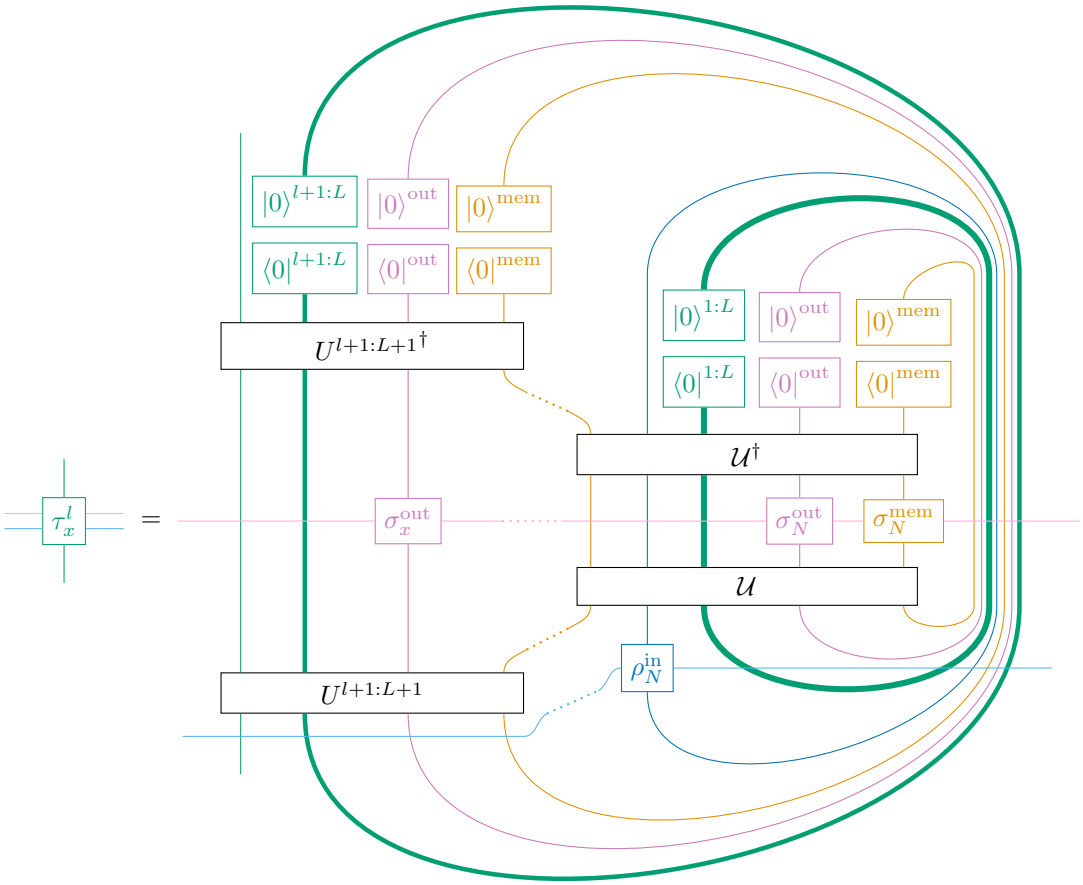
To get the cost or derivatives of the cost, we already contract the input legs of τ_x^0 with ρ_x^{in} and trace out the physical input legs. The remaining memory expression then is



(A.4.15)

which leaves us with open memory and virtual input and output legs.

We again iteratively insert the above equations into each other, starting from ν_x^l in Equation (A.4.14) and ending with the definition of ν_0^{mem} in Equation (A.4.12). This leads to

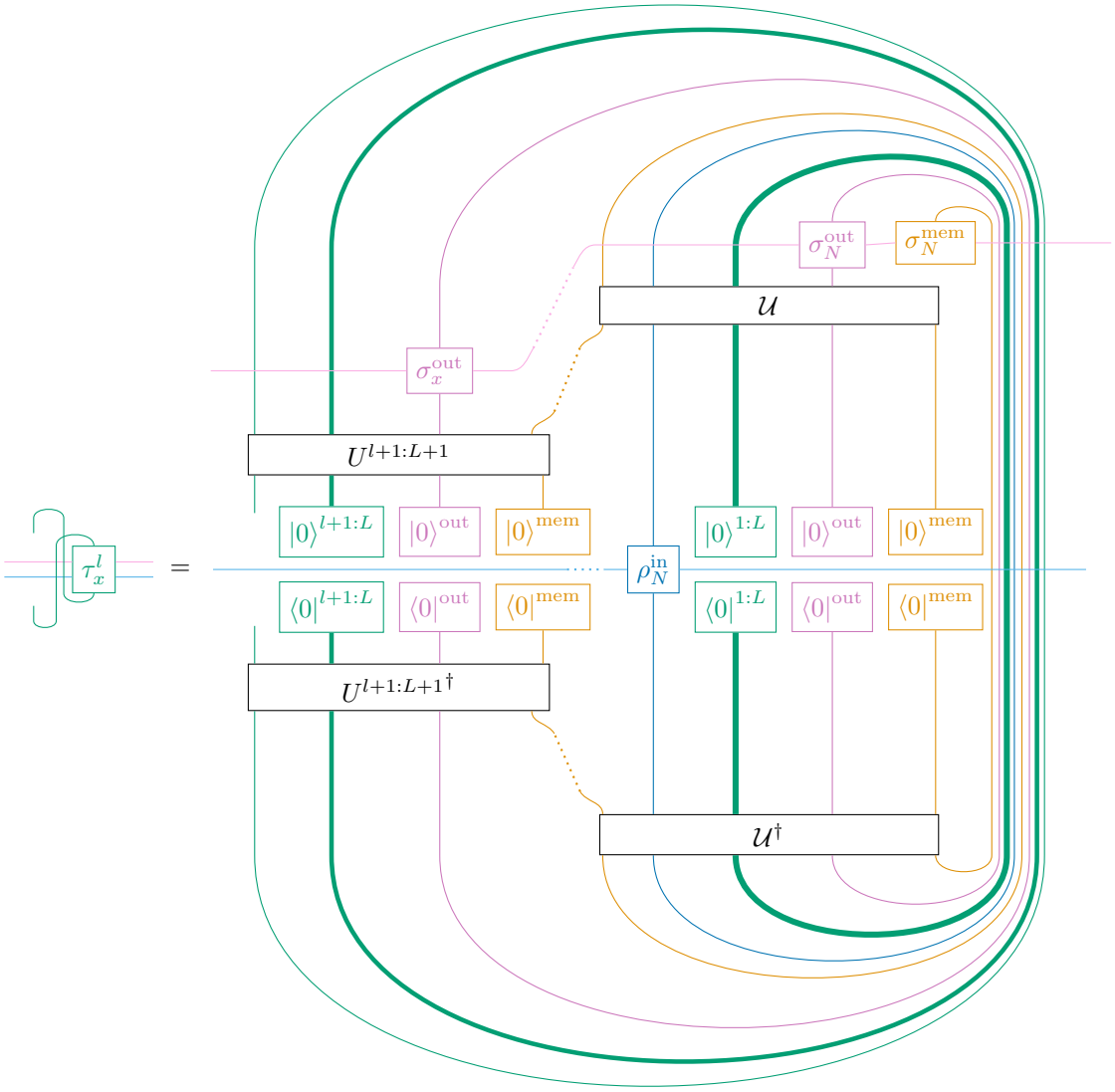


(A.4.16)

for $x = 1, ..., N$ and $l = 1, ..., L + 1$.

Rearranging this to have the legs facing inward, as we need it in this form for the derivative

of the cost, we get



(A.4.17)

A.4.3 Change of Cost function and Proof of Proposition 4.3

Again, we update our unitaries according to

$$U_j^l \mapsto U_j^{l\prime} = e^{i\epsilon K_j^l} U_j^l \tag{A.4.18}$$

for some small number ϵ. We have

$$\rho^{\text{OUT}} = \text{tr}_{1:N}^{0:L}\left(\mathcal{U}_N ... \mathcal{U}_1 \left(\rho^{\text{IN}} \otimes |0...0\rangle_{1:N}^{1:L+1} \langle 0...0|\right) \mathcal{U}_1^\dagger ... \mathcal{U}_N^\dagger\right)$$

and like before

$$\begin{aligned}
\rho^{\text{OUT}\prime} &= \rho^{\text{OUT}} + \frac{i\epsilon}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l} \text{tr}_{1:N}^{0:L}\Big(\mathcal{U}_{z+1:N} U_z^{l+1:L+1} U_{j+1:m_l\, z}^{l}\Big[K_{j\, z}^{l}, \\
&\quad U_{1:j\, z}^{l} U_z^{1:l-1} \mathcal{U}_{1:z-1}\big(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N}^{1:L+1} \langle 0\ldots0|\big)\mathcal{U}_{1:z-1}^\dagger U_z^{1:l-1\,\dagger} U_{1:j\, z}^{l\,\dagger}\Big] \\
&\quad U_{j+1:m_l\, z}^{l\,\dagger} U_z^{l+1:L+1\,\dagger} \mathcal{U}_{z+1:n}^\dagger\Big) + \mathcal{O}(\epsilon^2) \\
&= \rho^{\text{OUT}} + i\epsilon\delta\rho^{\text{OUT}} + \mathcal{O}(\epsilon^2)
\end{aligned}$$

which leads to

$$\begin{aligned}
\delta C &= \lim_{\epsilon\to 0}\frac{C' - C}{\epsilon} = \lim_{\epsilon\to 0}\frac{1 - \text{tr}\big(\rho^{\text{OUT}\prime}\sigma^{\text{OUT}}\big) - \big(1 - \text{tr}\big(\rho^{\text{OUT}}\sigma^{\text{OUT}}\big)\big)}{\epsilon} \\
&= -\lim_{\epsilon\to 0}\frac{i\epsilon\text{tr}\left(\left(\delta\rho^{\text{OUT}}\right)\sigma^{\text{OUT}}\right) + \mathcal{O}\left(\epsilon^2\right)}{\epsilon} = -i\text{tr}\left(\left(\delta\rho^{\text{OUT}}\right)\sigma^{\text{OUT}}\right) \\
&= -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\text{tr}\Big(\mathcal{U}_{z+1:N} U_z^{l+1:L+1} U_{j+1:m_l\, z}^{l}\Big[K_{j\, z}^{l}, U_{1:j\, z}^{l} U_z^{1:l-1}\mathcal{U}_{1:z-1} \\
&\quad \big(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N}^{1:L+1}\langle 0\ldots0|\big)\mathcal{U}_{1:z-1}^\dagger U_z^{1:l-1\,\dagger} U_{1:j\, z}^{l\,\dagger}\Big] U_{j+1:m_l\, z}^{l\,\dagger} U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:n}^\dagger \\
&\quad \Big(\sigma^{\text{OUT}} \otimes \mathbb{1}_{0:N}^{\text{mem}} \otimes \mathbb{1}_{1:N}^{\text{in},1:L}\Big)\Big) \\
&=: -\frac{i}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\delta C_{xj}^{l\,+} - \delta C_{xj}^{l\,+}
\end{aligned}$$

where

$$\begin{aligned}
C_{xj}^{l\,+} &= \text{tr}\Big(\mathcal{U}_{z+1:N} U_z^{l+1:L+1} U_{j+1:m_l\, z}^{l} K_{j\, z}^{l} U_{1:j\, z}^{l} U_z^{1:l-1}\mathcal{U}_{1:z-1}\big(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N}^{1:L+1}\langle 0\ldots0|\big) \\
&\quad \mathcal{U}_{1:z-1}^\dagger U_z^{1:l-1\,\dagger} U_{1:j\, z}^{l\,\dagger} U_{j+1:m_l\, z}^{l\,\dagger} U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:n}^\dagger\Big(\sigma^{\text{OUT}} \otimes \mathbb{1}_{0:N}^{\text{mem}} \otimes \mathbb{1}_{1:N}^{\text{in},1:L}\Big)\Big), \\
C_{xj}^{l\,-} &= \text{tr}\Big(\mathcal{U}_{z+1:N} U_z^{l+1:L+1} U_{j+1:m_l\, z}^{l} U_{1:j\, z}^{l} U_z^{1:l-1}\mathcal{U}_{1:z-1}\big(\rho^{\text{IN}} \otimes |0\ldots0\rangle_{1:N}^{1:L+1}\langle 0\ldots0|\big) \\
&\quad \mathcal{U}_{1:z-1}^\dagger U_z^{1:l-1\,\dagger} U_{1:j\, z}^{l\,\dagger} K_{j\, z}^{l} U_{j+1:m_l\, z}^{l\,\dagger} U_z^{l+1:L+1\,\dagger}\mathcal{U}_{z+1:n}^\dagger\Big(\sigma^{\text{OUT}} \otimes \mathbb{1}_{0:N}^{\text{mem}} \otimes \mathbb{1}_{1:N}^{\text{in},1:L}\Big)\Big).
\end{aligned}$$

In TNN, $\delta C_{x\, j}^{l\,+}$ and $\delta C_{x\, j}^{l\,-}$ are depicted in Figure A.1 and Figure A.2, respectively.

Plugging in the tensors we get from the feed-forward procedure (Equation (A.4.9)) and

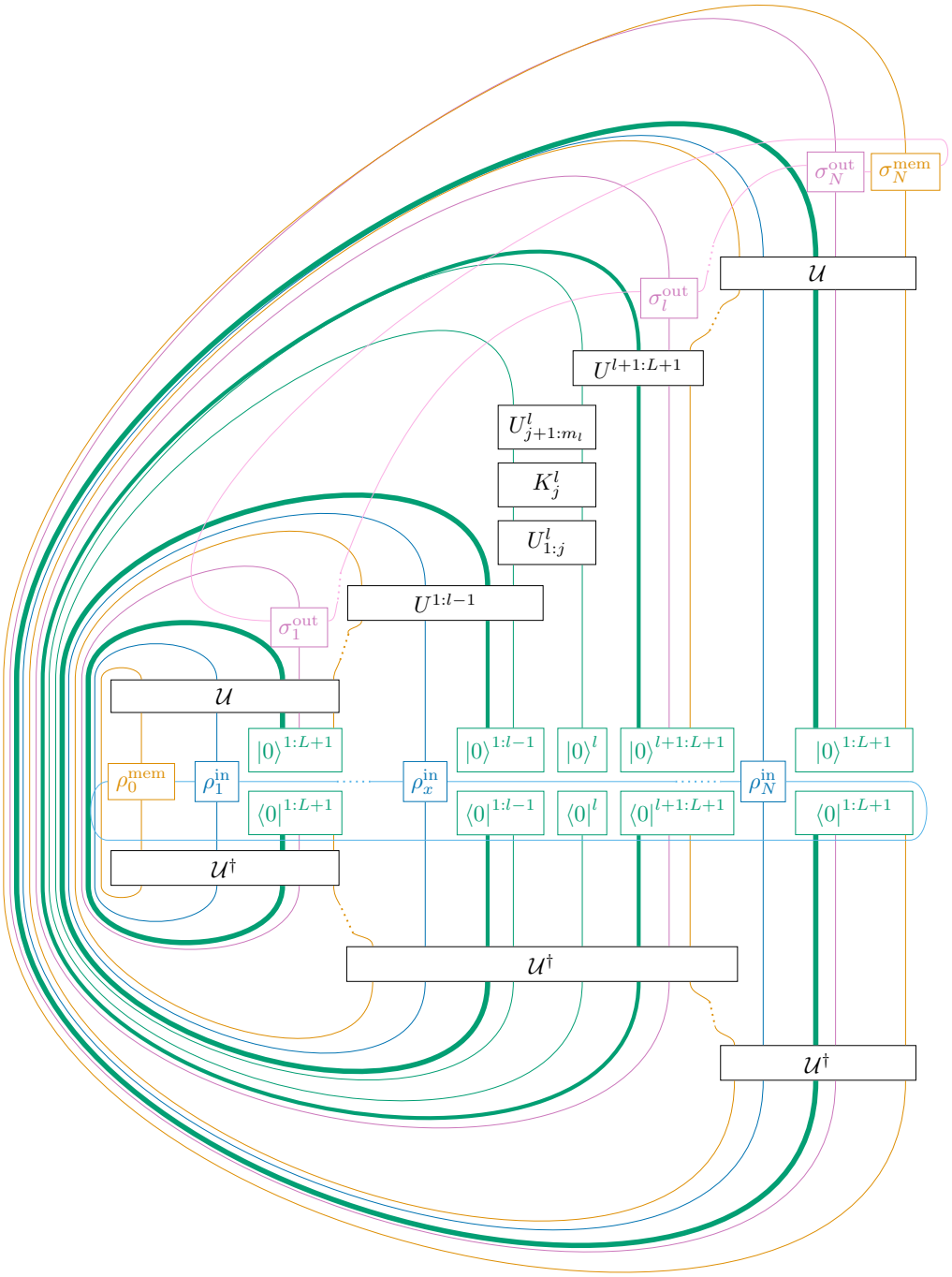


Figure A.1: ${\delta C^l_{x\,j}}^+$ for $j = 1, ..., m_l$, $l = 1, ..., L+1$, and $x = 1, ..., N_\alpha$ with training data as defined in Equations (A.4.1) and (A.4.2)

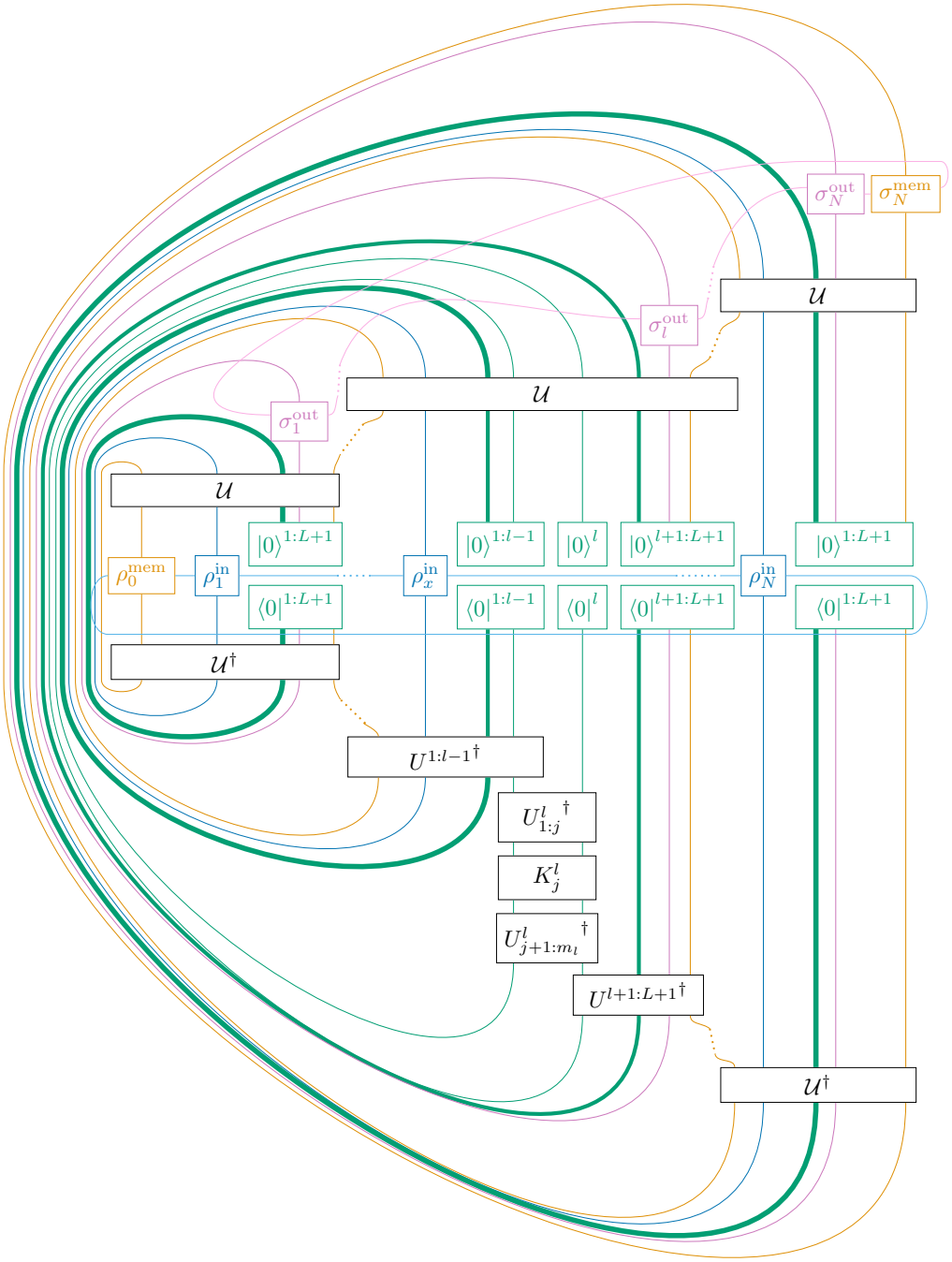
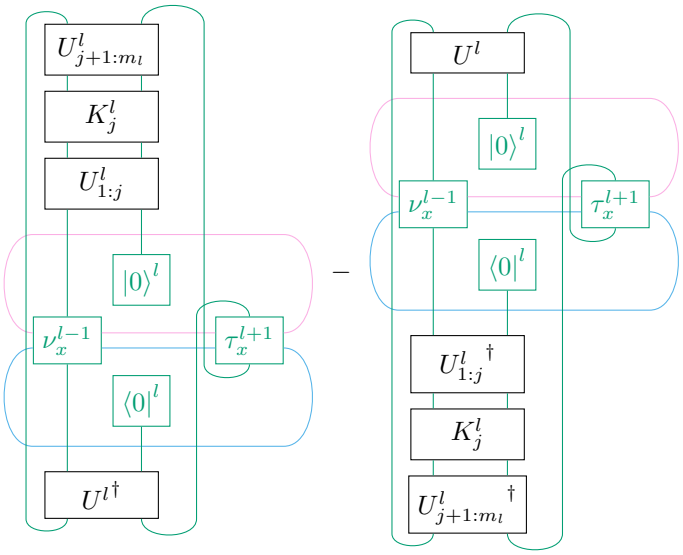


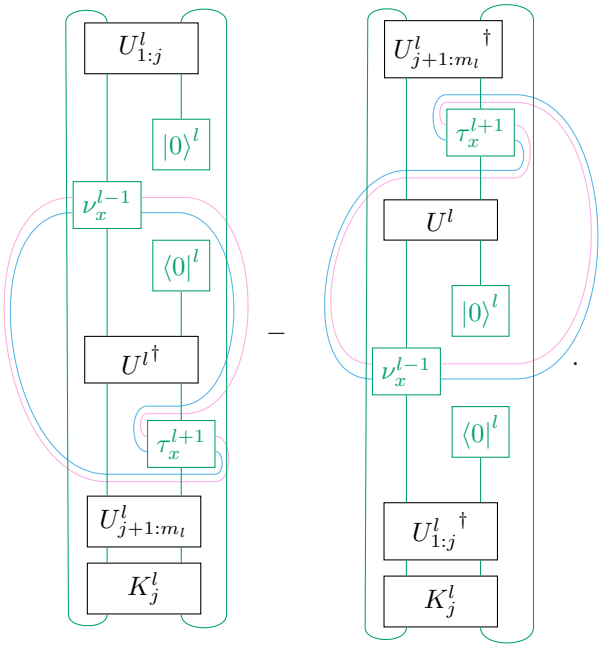
Figure A.2: $\delta C_{x\,j}^{l\;-}$ for $j = 1, ..., m_l$, $l = 1, ..., L+1$, and $x = 1, ..., N_\alpha$ with training data as defined in Equations (A.4.1) and (A.4.2)

backpropagation (Equation (A.4.17)), we get

$\delta C^l_{xj} =$ 

(A.4.19)

Rearranging and summing leads to

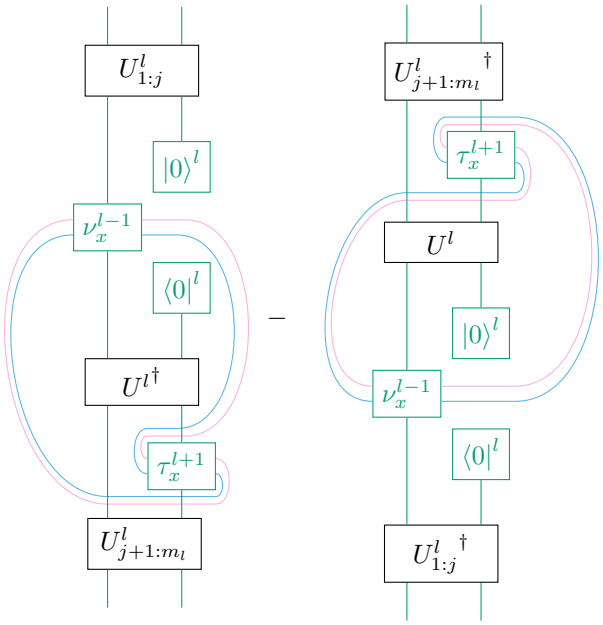
$\delta C = -\frac{i}{N} \sum_{x=1}^{N} \sum_{l=1}^{L+1} \sum_{j=1}^{m_l}$ 

.

(A.4.20)

Setting

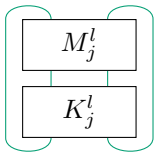
$$M_j^l = -\sum_{x=1}^{N}$$



(A.4.21)

leads to

$$\delta C = -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}$$



$$= -i\sum_{l=1}^{L+1}\sum_{j=1}^{m_l}\operatorname{tr}(M_j^l K_j^l)$$

(A.4.22)

which concludes the proof of Proposition 4.3 and is again minimized (if we assume K_j^l to be bounded) by

$$K_j^l = \frac{2^{m_{l-1}}i}{\lambda}\operatorname{tr}^l_{1:j-1}\Big(\operatorname{tr}^l_{j+1:m_l}\left(M_j^l\right)\Big).$$

B

Additional Information on RL for Fiber Coupling

B.1 Hyperparameters of RL Algorithms

We use the standard hyperparameters in StableBaselines3 and sb3-contrib, version 2.3.0 [340]. Nevertheless, we show them here. parameters printed in bold are the parameters that are not the default parameters but appear like that in their tutorials.

TQC learning rate: 0.0003, replay buffer size: 1000000, learning starts after 100 steps, batch size: 256, soft update coefficient: 0.005, discount factor: 0.99, update model every step, do 1 gradient step after each rollout, no added action noise, update target network every 1 step, number of quantiles to drop per net: 2, number of critics networks: 2, number of quantiles for critic: 25

SAC learning rate: 0.0003, replay buffer size: 1000000, learning starts after 100 steps, batch size: 256, soft update coefficient: 0.005, discount factor: 0.99, update model every step, do 1 gradient step after each rollout, no added action noise, update target network every 1 step,

TD3 learning rate: 0.001, replay buffer size: 1000000, learning starts after 100 steps, batch size: 256, soft update coefficient: 0.005, discount factor: 0.99, update model every step, do 1 gradient step after each rollout, **action noise: NormalActionNoise(mean=np.zeros(number actions), sigma=0.1 × np.ones(number actions)**, policy and target network updated every 2 steps, standard deviation of smoothing noise on target policy: 0.2, clip absolute value of target policy smoothing noise at: 0.5

DDPG learning rate: 0.001, replay buffer size: 1000000, learning starts after 100 steps, batch size: 256, soft update coefficient: 0.005, discount factor: 0.99, update model every step, do 1 gradient step after each rollout, **action noise: NormalActionNoise(mean=np.zeros(number actions), sigma=0.1 × np.ones(number actions)**

PPO learning rate: 0.0003, number of steps between updates: 2048, batch size: 64, number of epochs when optimizing surrogate loss: 10, discount factor: 0.99, factor for trade-off

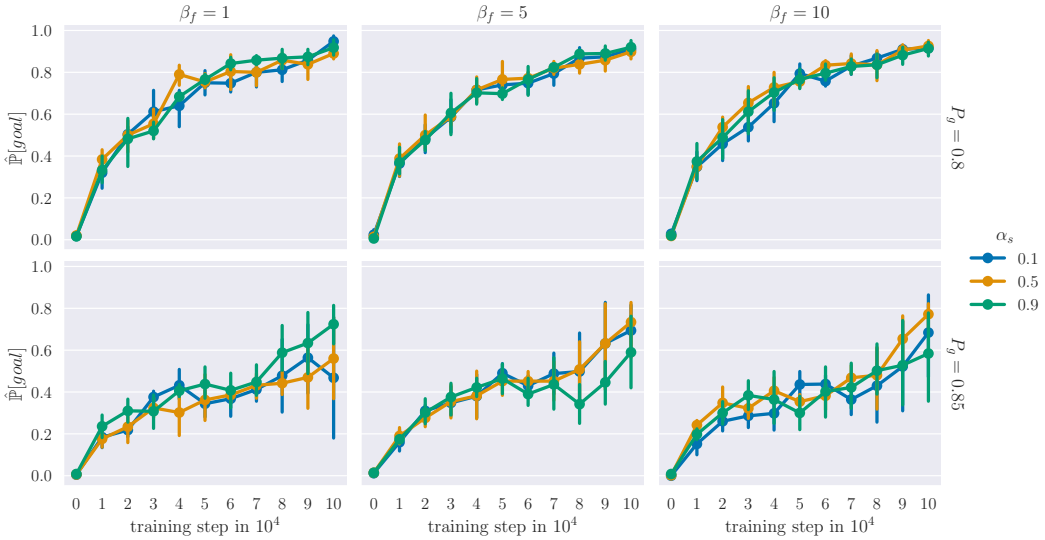


Figure B.1: **Step reward tuning results:** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different fail reward parameters α_s, β_s using the parameters in Table 5.1, $P_{\text{goal}} = 0.8, 0.85$ and TQC.

between bias vs. variance for GAE: 0.95, clip range: 0.2, normalize advantage, entropy coefficient: 0.0, value function coefficient for loss calculation: 0.5, maximum norm for gradient clipping: 0.5

A2C learning rate: 0.0007, number of steps between updates: 5, discount factor: 0.99, factor for trade-off between bias vs. variance for GAE: 1.0, entropy coefficient: 0.0, value function coefficient for loss calculation: 0.5, maximum norm for gradient clipping: 0.5, RMSProp epsilon: 1e-05, use RMSprop

B.2 Reward Hyperparameters

B.2.1 Step Reward

We tested different α_s, β_s using $P_{\text{goal}} = 0.8, 0.85$. The results are shown in Figure B.1. For $P_{\text{goal}} = 0.8$, there is not much of a difference between the different values, but $\alpha_s = 0.9$ performs slightly better than the other two. For $P_{\text{goal}} = 0.85$, the differences are a little more pronounced, but the standard deviation is generally higher. We went with $\alpha_s = 0.9, \beta_s = 5$ in the other experiments.

B.2.2 Goal Reward

We tested different $\alpha_g, \beta_{g1}, \beta_{g2}$. The results are shown in Figure B.2. We can clearly see that we should choose $\beta_{g2} = 1$ and not $\alpha_g = 0.9$. For the other experiments, we went with $\beta_{g1} = 5, \alpha_g = 0.5$ as this curve seems to be still rising in the end.

B.2.3 Fail Reward

We tested different $\alpha_f, \beta_{f1}, \beta_{f2}$. The results are shown in Figure B.3. Again, the choice

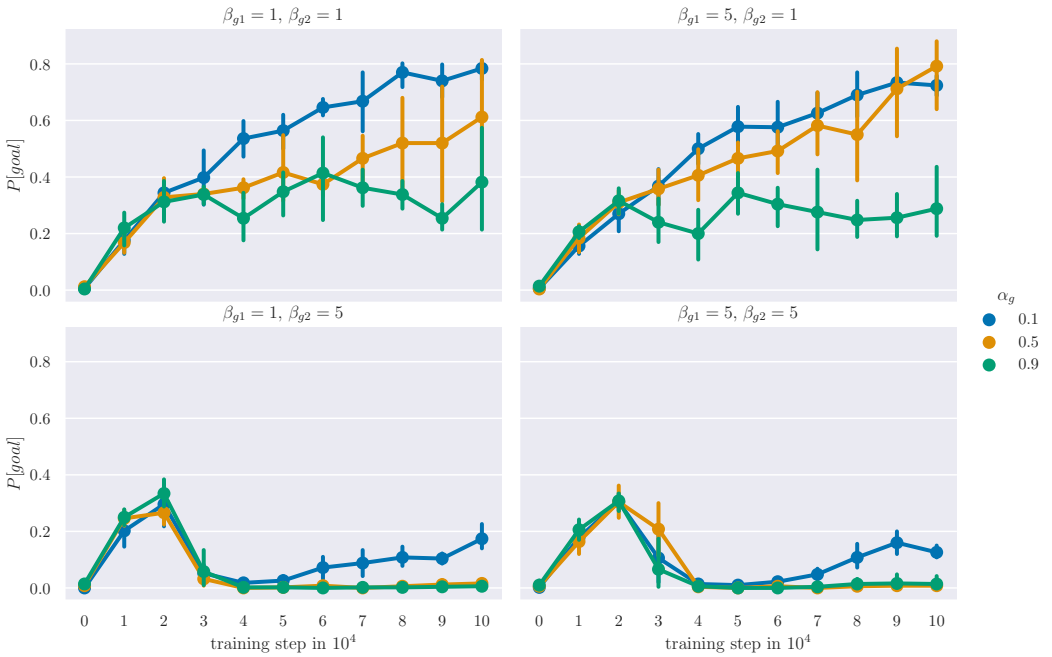


Figure B.2: **Goal reward tuning results:** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different goal reward parameters $\alpha_g, \beta_{g1}, \beta_{g2}$ using the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

between the different parameters is not very clear. There is a small tendency towards bigger β and not using $\alpha = 0.1$. Because we see both reward parts as equally important, we use $\alpha_f = 0.5, \beta_{f1} = 5, \beta_{f2} = 5$.

B.2.4 Reducing the Step Reward over Time

We also tested reducing the pre-factor A_s of the step reward over the course of training for $P_{\text{goal}} = 0.8, 0.9$. The results are shown in Figure B.4.

For $P_{\text{goal}} = 0.8$, the drop suffered with $A_s = 10$ is not as bad as for $P_{\text{goal}} = 0.85$, and the probability of reaching the goal is near one at the end of training, even with $A_s = 100$. Nevertheless, reducing A_s from 100 to 10 over the course of training significantly helps.

This is a different story for $P_{\text{goal}} = 0.9$. Overall, $A_s = 10$ performs best if one wants to reach the goal with nearly 100% probability as quickly as possible. If one, however, only needs to reach the goal in around 60% of episodes, reducing A_s from 100 to 10 over the course of $2 \cdot 10^5 - 4 \cdot 10^5$ training steps can be quite helpful. Choosing a constant $A_s = 100$ is worse than for the other cases. Although there is a small boost in performance at the start, the probability of reaching the goal quickly goes down to nearly 0 over the course of training.

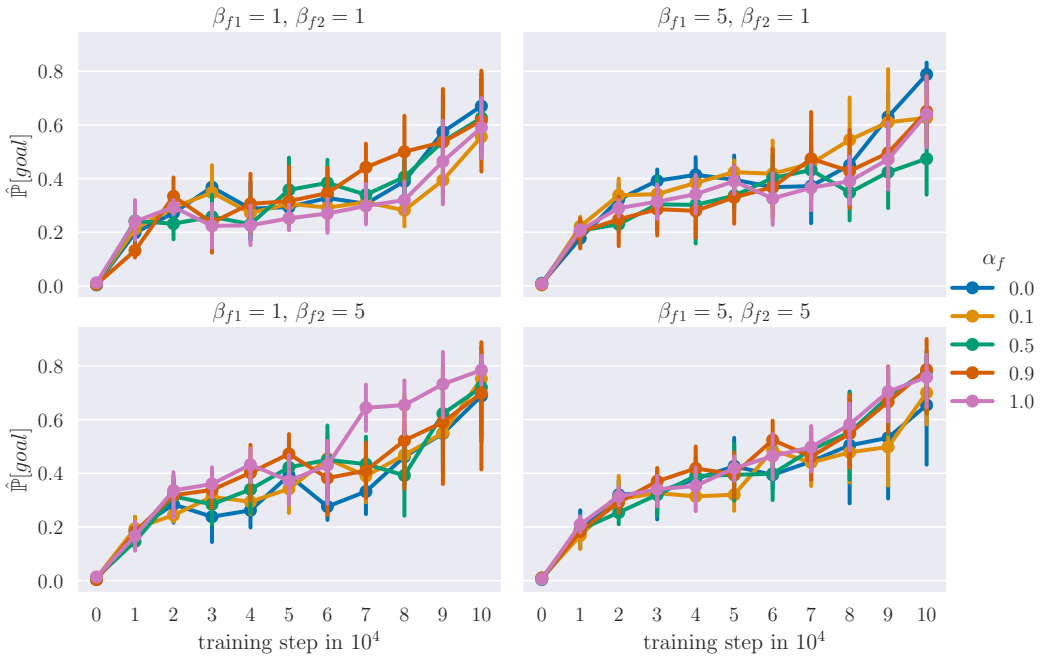


Figure B.3: **Fail reward tuning results:** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different fail reward parameters $\alpha_f, \beta_{f1}, \beta_{f2}$ using the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.85$ and TQC. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

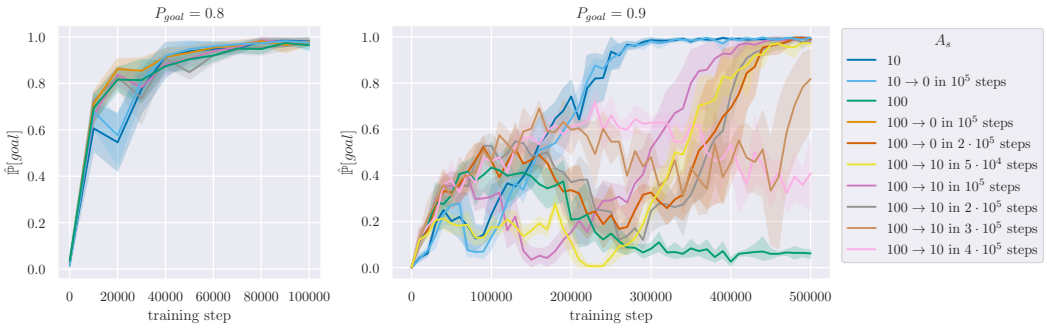


Figure B.4: **Reducing A_s over the course of training:** We show the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different pre-factors of the step reward that changes over the course of training, i.e., A_s is decreased linearly from a starting value of $10, 100$ to an ending value of $0, 10$ over the course of the first $5 \cdot 10^4, ..., 4 \cdot 10^5$ training steps using the parameters in Tables 5.1 and 5.2, $P_{\text{goal}} = 0.8, 0.9$ and TQC. The error bands have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

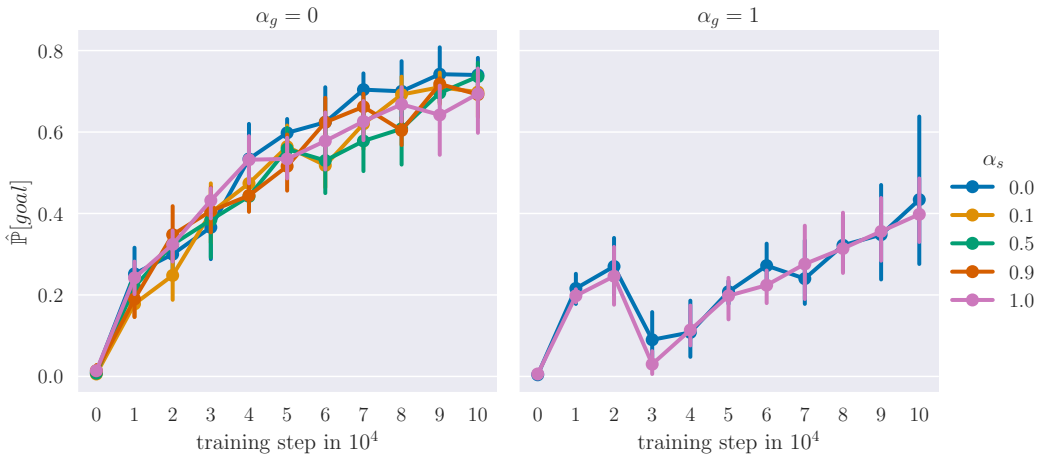


Figure B.5: **Tuning of simplified reward:** We plotted the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different $\alpha_g = 0, 1$, and $\alpha_s = 0, 0.1, 0.5, 0.9, 1$ for the simplified reward r_{simpler} using the parameters in Tables 5.1 and 5.2, $A_f = 10$, $P_{\text{goal}} = 0.85$ and TQC. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

B.3 Simplified Reward Functions

The reward we use in the main part

$$r_{\text{standard},t} = \begin{cases} -A_f \left((1-\alpha_f) \exp\left(-\beta_{f,1} \frac{t}{T}\right) + \alpha_f \exp\left(-\beta_{f,2} \frac{P_t}{P_{\text{fail}}}\right) \right) & \text{if } P_t < P_{\text{fail}} \\ A_g \left((1-\alpha_g) \exp\left(-\beta_{g,1} \frac{t}{T}\right) + \alpha_g \exp\left(\beta_{g,2} \frac{P_t}{P_{\text{goal}}}\right) \right) & \text{if } P_t > P_{\text{goal}} \\ \frac{A_s}{T} \left((1-\alpha_s) \exp\left(\beta_s (P_t - P_{\text{goal}})\right) + \alpha_s \left(P_t - P_{\text{min}}\right) \right) & \text{else} \end{cases}$$

is comparatively complex. Starting from there, we can try to simplify it, e.g., by setting one or more of the α's to 0 or 1 or by choosing constant rewards for one or more of the reward parts. We choose to try a constant fail reward, which leads to

$$r_{\text{simpler},t} = \begin{cases} -A_f & \text{if } P_t < P_{\text{fail}} \\ A_g \left((1-\alpha_g) \exp\left(-\beta_{g,1} \frac{t}{T}\right) + \alpha_g \exp\left(\beta_{g,2} \frac{P_t}{P_{\text{goal}}}\right) \right) & \text{if } P_t > P_{\text{goal}} \\ \frac{A_s}{T} \left((1-\alpha_s) \exp\left(\beta_s (P_t - P_{\text{goal}})\right) + \alpha_s \left(P_t - P_{\text{min}}\right) \right) & \text{else} \end{cases} .$$

For $A_f = 10$, and $P_{\text{goal}} = 0.85$, we test different values for $\alpha_s = 0, 0.1, 0.5, 0.9, 1$ and $\alpha_g = 0, 1$ using TQC. The results are shown in Figure B.5. The agent trained with $\alpha_g = 0$ clearly outperforms the one with $\alpha_g = 1$. The results for the different α_s' are not so obvious. However, the one trained with $\alpha_s = 0$ slightly surpasses the other candidates. These values lead to the simplified reward

$$r_{\text{simplified},t} = \begin{cases} -A_f & \text{if } P_t < P_{\text{fail}} \\ A_g \exp\left(-\beta_{g,1} \frac{t}{T}\right) & \text{if } P_t > P_{\text{goal}} \\ \frac{A_s}{T} \exp\left(\beta_s (P_t - P_{\text{goal}})\right) & \text{else} \end{cases} .$$

Furthermore, we compare this simplified reward to the standard reward and the rewards presented in Appendix B.4. Figure B.6 not only shows the probability of reaching the

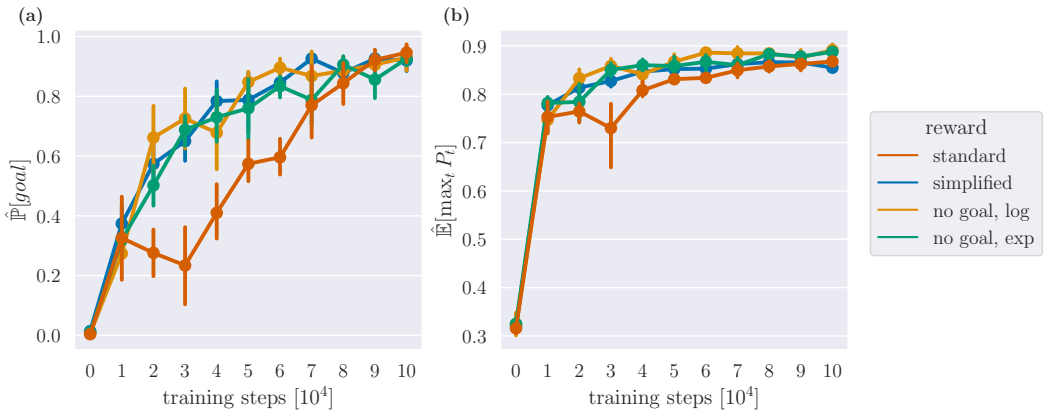


Figure B.6: **Comparison of standard, simplified, and no goal rewards.** Panel (a) shows the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for different rewards using the parameters in Table 5.1, $P_{\text{goal}} = 0.85$ and TQC. Panel (b) shows the average of the maximum power in each tested episode $\hat{\mathbb{E}}[\max_t P_t]$ against the training step for the same rewards and parameters. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

goal $\hat{\mathbb{P}}[goal]$ but also the average of the maximum power in each tested episode $\hat{\mathbb{E}}[\max_t P_t]$ against the training step. The simplified reward clearly outperforms the standard reward and performs approximately as well as the presented rewards for training an agent without a goal.

B.4 Training the Agent without a Fixed Goal

We now shortly look at the task of learning to optimize the power without setting a goal. We hence have to make the following modifications to our environment: The episodes do not end if $P > P_{\text{goal}}$, and no reward is associated with reaching the goal.

We test the following reward functions: Firstly, we test a reward function similar to our "simpler" reward function without the goal reward

$$r_{\text{no goal, exp+lin},t} = \begin{cases} -A_f & \text{if } P_t < P_{\text{fail}} \\ \frac{A_s}{T}\left((1-\alpha_s)\exp\left(\beta_s(P_t - P_{\text{goal}})\right) + \alpha_s\left(P_t - P_{\text{min}}\right)\right) & \text{else} \end{cases}$$

with $A_f = 10$, $A_s = 10$, $\beta_s = 5$, and $\alpha_s = 0, 0.5, 1$. Secondly, we test a reward function inspired by the work [325]

$$r_{\text{no goal, log},t} = \begin{cases} -A_f & \text{if } P_t < P_{\text{fail}} \\ \frac{A_s}{T}\left(\frac{P_t}{A+\epsilon} - 1 - \log\left(1 - \frac{P_t}{A+\epsilon}\right)\right) & \text{else} \end{cases}$$

where $A = 0.92$ is the amplitude of the Gaussian we used, $\epsilon = 10^{-6}$ is a small number used in case $P_t = A$, $A_s = 10$, and $A_f = 10, 50$.

Although the episodes do not end, when the agent reaches the goal, we can nevertheless test them on their ability to reach a goal power, as this is important for the experimental use of

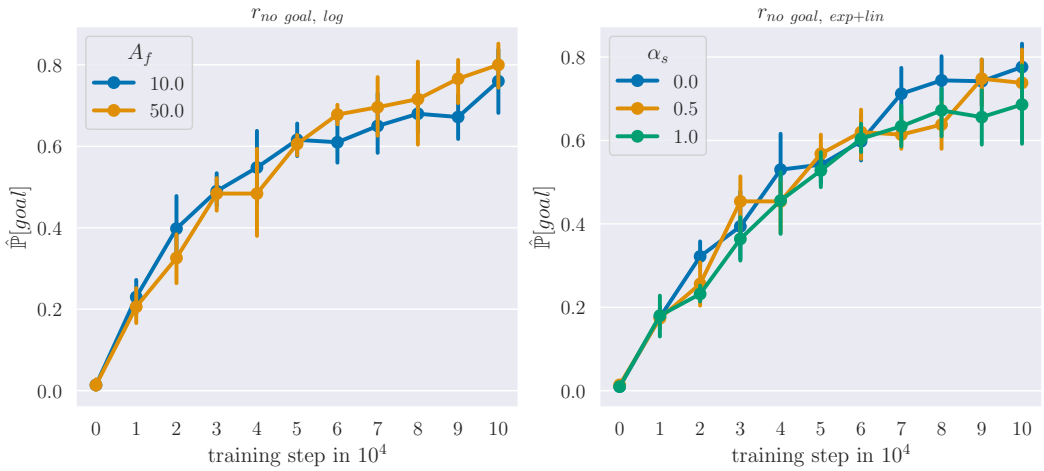


Figure B.7: **Different rewards for training an agent without setting a goal.** The plots show the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step for training the agent without a goal with different rewards, parameters in Tables 5.1, $T = 20$, $P_{\text{goal}} = 0.85$ and TQC. For $r_{\text{no goal, exp+lin}}$, we use $A_f = 10$, $A_s = 10$, $\beta_s = 5$, and $\alpha_s = 0, 0.5, 1$, and for $r_{\text{no goal, log}}$, we use $P_t = A$, $A_s = 10$, and $A_f = 10, 50$. The error bars have a size of 2σ in each direction, where σ is the standard deviation of the five tests.

the agent. Figure B.7 shows the probability of reaching the goal $\hat{\mathbb{P}}[goal]$ against the training step. The differences in performance for the different reward parameters are only minor. Still, the agents trained with $\alpha_s = 0$ for $r_{\text{no goal, exp+lin}}$ and $A_f = 50$ for $r_{\text{no goal, log},t}$ slightly outperform the ones trained with the other parameters. Hence, $r_{\text{no goal, exp+lin}}$ simplifies to

$$r_{\text{no goal, exp},t} = \begin{cases} -A_f & \text{if } P_t < P_{\text{fail}} \\ \frac{A_s}{T} \exp\left(\beta_s (P_t - P_{\text{goal}})\right) & \text{else} \end{cases} .$$

See Figure B.6 for a direct comparison of $r_{\text{no goal, exp}}$, $r_{\text{no goal, log}}$, $r_{\text{simplified}}$, and r_{standard}. Panel (a) shows the probability of reaching a goal of $P_{\text{goal}} = 0.85$, and Panel (b) shows the average of the maximum power reached per episode. For both figures of merit, the standard reward function is outperformed by the others. The performance of the other three is comparable regarding the probability of reaching the goal, but $r_{\text{no goal, exp}}$ performs slightly worse. However, if we consider $\hat{\mathbb{E}}[\max_t P_t]$ instead, the agents trained without a goal outperform the ones trained with a goal.

In total, it can be beneficial to train the agent without a goal, even if it will be applied with a goal. The rewards $r_{\text{no goal, exp}}$ and $r_{\text{no goal, log}}$ perform similar, although $r_{\text{no goal, log}}$ slightly outperforms the other. However, to work with $r_{\text{no goal, log}}$ that well, we need a good estimate for the maximum possible power. In the virtual testbed, this is easy to get by just taking the amplitude. In the lab, on the other hand, this would not be as easy, which is why we suggest $r_{\text{no goal, exp}}$ for training in the experiment without a goal.

Acronyms

A2C advantage actor-critic. 22, 28, 29, 108, 120

A3C asynchronous advantage actor-critic. 29

AI artificial intelligence. 1, 17

CC ML machine learning with classical data and classical algorithms. 2, 51, 117

CPI conservative policy iteration. 30

CQ ML machine learning with classical data and quantum algorithms. 2, 51, 52, 57, 59, 117

cv continuous variable. 52, 59

DDPG deep deterministic policy gradient. 30–32, 92, 108, 120

DQN deep Q-network. 22, 23, 26, 27, 29–31, 33

DQNN dissipative quantum neural network. iii, 2–4, 36, 52–57, 59–63, 66–70, 72, 73, 75, 76, 78, 79, 81, 83–89, 117–119, 132–134, 136, 143, 146, 149

DQRNN dissipative quantum recurrent neural network. iii, 2–4, 59–63, 65–68, 70, 72–74, 78, 79, 81, 83–89, 117–119, 121, 132, 143

EEG electroencephalography. 62

EMG electromyography. 62

fc fully connected. 10, 11, 14, 53, 54, 61, 119

ff feed-forward. 10, 11, 13, 53, 54, 56, 59–61, 66, 67, 81, 83–89, 94, 117–119, 133, 134, 136

GHZ Greenberger-Horne-Zeilinger. 49, 50

GRU gated recurrent unit. 13, 15, 62

HER hindsight experience replay. 27

i.i.d. independent and identically distributed. 2, 6, 59, 117

iff if and only if. 6, 18, 20, 21, 36, 37, 40, 42–44

KL Kullback–Leibler. 7, 30, 32

LSTM long short-term memory. 13, 15, 62

MAE mean absolute error. 7

MC Monte-Carlo. 24, 25

MDP Markov decision process. 17–28, 114

ML machine learning. iii, 1–5, 8, 12, 16, 45, 51, 52, 57, 91, 94, 117, 121

MPO matrix product operator. 50, 72, 143

MPS matrix product state. 49, 50, 79, 118

MSE mean squared error. 7

NISQ noisy intermediate-scale quantum. vii, 51, 55, 59, 65, 66, 88

NLP natural language processing. 2, 13

NN neural network. 1–4, 6, 8, 10–16, 22, 26–30, 51–55, 60–62, 66, 68, 70, 89, 94, 105, 117–119, 132

NNs neural networks. 11

ONB orthonormal basis. 36, 37, 41

PER prioritized experience replay. 27

PID proportional–integral–derivative. 114

POMDP partially observable Markov decision process. 20, 21, 93, 95, 119

POVM positive operator valued measurements. 39, 42

PPO proximal policy optimization. 22, 23, 30, 91, 92, 108, 120

QC quantum computer. 1, 3, 35, 36

QC quantum computing. 2, 3, 35, 52, 59, 66

QC ML machine learning with quantum data and classical algorithms. 2, 51, 117

QCNN quantum convolutional neural network. 52

QEC quantum error correction. 1, 2, 51, 52

QGAN quantum generative adversarial network. 52

QI quantum information. 2, 4, 35, 37, 51, 117, 119

QML quantum machine learning. 2, 4, 51, 57, 117

QNN quantum neural network. 2, 3, 52, 57, 62

QOMDP quantum observable Markov decision process. 119

QQ ML machine learning with quantum data and quantum algorithms. 2, 51, 52, 57, 59, 117

QRC quantum reservoir computing. 62, 63

QRNN quantum recurrent neural network. iii, 2, 4, 59, 62, 63, 117

RL reinforcement learning. iii, 1, 3–5, 8, 16, 17, 22, 51, 52, 91, 92, 94, 95, 103, 108, 113–115, 117, 119–121

RNN recurrent neural network. iii, 2, 3, 13–15, 44, 59–62, 89, 117–119

SAC soft actor-critic. iii, 3, 22, 23, 32, 33, 92, 108–110, 113, 114, 120

SL supervised learning. 1, 2, 4–8, 16, 17, 51, 52, 117

SVD singular value decomposition. 48, 49

TD temporal difference. 25, 26

TD3 twin delayed deep deterministic policy gradient. 3, 23, 31, 32, 92, 108, 114, 120

TN tensor network. 4, 48, 78, 89, 117, 118

TNN tensor network notation. 46–48, 72, 145, 146, 152

TQC truncated quantile critics. iii, 3, 32, 34, 92, 99–104, 106–114, 120, 157–163

TRPO trust region policy optimization. 29, 30

w.l.o.g. without loss of generality. 41, 60

w.r.t. with respect to. 6, 12, 22, 23, 25, 30, 31, 36, 37, 41, 46, 56, 59, 63, 67, 71, 79–86, 118, 129, 131, 133, 142

Bibliography

[1] OpenAI. Dall-e 3. URL https://openai.com/index/dall-e-3/.

[2] Microsoft. Copilot. URL https://copilot.microsoft.com/.

[3] Adobe. Adobe firefy. URL https://firefly.adobe.com/.

[4] GitHub. Copilot. URL https://github.com/features/copilot.

[5] Capes, T. *et al.* Siri on-device deep learning-guided unit selection text-to-speech system. In *Interspeech 2017*, 4011–4015 (ISCA, 2017). URL https://www.isca-archive.org/interspeech_2017/capes17_interspeech.html.

[6] OpenAI. Chatgpt. URL https://chat.openai.com.

[7] Brown, T. *et al.* Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. & Lin, H. (eds.) *Advances in Neural Information Processing Systems*, vol. 33, 1877–1901 (Curran Associates, Inc., 2020). URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[8] DeepL. Deepl translator. URL https://www.deepl.com/.

[9] Beel, J., Langer, S., Nürnberger, A. & Genzmehr, M. The impact of demographics (age and gender) and other user-characteristics on evaluating recommender systems. In Aalberg, T., Papatheodorou, C., Dobreva, M., Tsakonas, G. & Farrugia, C. J. (eds.) *Research and Advanced Technology for Digital Libraries*, 396–400 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013). URL https://doi.org/10.1007/978-3-642-40501-3_45.

[10] Gomez-Uribe, C. A. & Hunt, N. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.* **6** (2015). URL https://doi.org/10.1145/2843948.

[11] Fanca, A., Puscasiu, A., Gota, D.-I. & Valean, H. Recommendation systems with machine learning. In *2020 21th International Carpathian Control Conference (ICCC)* (2020). URL https://ieeexplore.ieee.org/abstract/document/9257290.

[12] Lau, J. Google maps 101: How ai helps predict traffic and determine routes (2020). URL https://blog.google/products/maps/google-maps-101-how-ai-helps-predict-traffic-and-determine-routes/.

[13] Gupta, U. *et al.* The architectural implications of facebook's dnn-based personalized recommendation (2020). URL https://arxiv.org/abs/1906.03109. 1906.03109.

[14] Purba, K. R., Asirvatham, D. & Murugesan, R. K. Classification of instagram fake users using supervised machine learning algorithms. *International Journal of Electrical and Computer Engineering* **10**, 2763 (2020). URL http://doi.org/10.11591/ijece.v10i3.pp2763-2772.

[15] How instagram uses artificial intelligence to moderate content. Instagram Help Center. URL https://help.instagram.com/423837189385631/?helpref=uf_share.

[16] Awoyemi, J. O., Adetunmbi, A. O. & Oluwadare, S. A. Credit card fraud detection using machine learning techniques: A comparative analysis. In *2017 International Conference on Computing Networking and Informatics (ICCNI)* (IEEE, 2017). URL https://ieeexplore.ieee.org/abstract/document/8123782.

[17] Dornadula, V. N. & Geetha, S. Credit card fraud detection using machine learning algorithms. *Procedia Computer Science* **165**, 631–641 (2019). URL https://doi.org/10.1016/j.procs.2020.01.057.

[18] Dastile, X., Celik, T. & Potsane, M. Statistical and machine learning models in credit scoring: A systematic literature survey. *Applied Soft Computing* **91**, 106263 (2020). URL https://doi.org/10.1016/j.asoc.2020.106263.

[19] Teles, G., Rodrigues, J. J. P. C., Saleem, K., Kozlov, S. & Rabêlo, R. A. L. Machine learning and decision support system on credit scoring. *Neural Computing and Applications* **32**, 9809–9826 (2019). URL https://doi.org/10.1007/s00521-019-04537-7.

[20] Regulation (eu) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence and amending regulations (ec) no 300/2008, (eu) no 167/2013, (eu) no 168/2013, (eu) 2018/858, (eu) 2018/1139 and (eu) 2019/2144 and directives 2014/90/eu, (eu) 2016/797 and (eu) 2020/1828 (artificial intelligence act) (text with eea relevance). OJ L (2024). URL http://data.europa.eu/eli/reg/2024/1689/oj.

[21] Grimmer, J., Roberts, M. E. & Stewart, B. M. Machine learning for social science: An agnostic approach. *Annual Review of Political Science* **24**, 395–419 (2021). URL https://doi.org/10.1146/annurev-polisci-053119-015921.

[22] de Slegte, J., Van Droogenbroeck, F., Spruyt, B., Verboven, S. & Ginis, V. The use of machine learning methods in political science: An in-depth literature review. *Political Studies Review* (2024). URL https://doi.org/10.1177/14789299241265084.

[23] Molina, M. & Garip, F. Machine learning for sociology. *Annual Review of Sociology* **45**, 27–45 (2019). URL https://doi.org/10.1146/annurev-soc-073117-041106.

[24] Deo, R. C. Machine learning in medicine. *Circulation* **132**, 1920–1930 (2015). URL https://doi.org/10.1161/CIRCULATIONAHA.115.001593.

[25] MacEachern, S. J. & Forkert, N. D. Machine learning for precision medicine. *Genome* **64**, 416–425 (2020). URL https://doi.org/10.1139/gen-2020-0131.

[26] Cleophas, T. J. & Zwinderman, A. H. *Machine Learning in Medicine* (Springer Dordrecht, 2013), 1 edn. URL https://link.springer.com/book/10.1007/978-94-007-5824-7.

[27] Larrañaga, P. *et al.* Machine learning in bioinformatics. *Briefings in Bioinformatics* **7**, 86–112 (2006). URL https://doi.org/10.1093/bib/bbk007.

[28] Schölkopf, B., Tsuda, K. & Vert, J.-P. (eds.) *Kernel Methods in Computational Biology* (The MIT Press, 2004). URL https://doi.org/10.7551/mitpress/4057.001.0001.

[29] Keith, J. A. *et al.* Combining machine learning and computational chemistry for predictive insights into chemical systems. *Chem. Rev.* **121**, 9816–9872 (2021). URL https://doi.org/10.1021/acs.chemrev.1c00107.

[30] Dral, P. O. Quantum chemistry in the age of machine learning. *The Journal of Physical Chemistry Letters* **11**, 2336–2347 (2020). URL https://doi.org/10.1021/acs.jpclett.9b03664.

[31] Meuwly, M. Machine learning for chemical reactions. *Chemical Reviews* **121**, 10218–10239 (2021). URL https://doi.org/10.1021/acs.chemrev.1c00033.

[32] Karniadakis, G. E. *et al.* Physics-informed machine learning. *Nature Reviews Physics* 422–440 (2021). URL https://doi.org/10.1038/s42254-021-00314-5.

[33] Karagiorgi, G., Kasieczka, G., Kravitz, S., Nachman, B. & Shih, D. Machine learning in the search for new fundamental physics. *Nature Reviews Physics* **4**, 399–412 (2022). URL https://doi.org/10.1038/s42254-022-00455-1.

[34] Carleo, G. *et al.* Machine learning and the physical sciences. *Reviews of Modern Physics* **91**, 045002 (2019). URL https://link.aps.org/doi/10.1103/RevModPhys.91.045002.

[35] 2024, N. P. O. A. Press Release (2024). URL https://www.nobelprize.org/prizes/physics/2024/press-release/.

[36] 2024, N. P. O. A. Press release (2024). URL https://www.nobelprize.org/prizes/chemistry/2024/press-release/.

[37] Bishop, C. M. *Pattern Recognition and Machine Learning* (Springer Cham, 2006). URL https://doi.org/10.1007/978-3-030-95995-1.

[38] Shalev-Shwartz, S. & Ben-David, S. *Understanding Machine Learning* (Cambridge University Press, 2014). URL https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/.

[39] Sidey-Gibbons, J. A. M. & Sidey-Gibbons, C. J. Machine learning in medicine: a practical introduction. *BMC Medical Research Methodology* **19** (2019). URL https://doi.org/10.1186/s12874-019-0681-4.

[40] Amrane, M., Oukid, S., Gagaoua, I. & Ensarİ, T. Breast cancer classification using machine learning. In *2018 Electric Electronics, Computer Science, Biomedical Engineerings' Meeting (EBBT)* (2018). URL https://ieeexplore.ieee.org/abstract/document/8391453.

[41] Varsamopoulos, S., Criger, B. & Bertels, K. Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology* **3**, 015004 (2017). URL https://iopscience.iop.org/article/10.1088/2058-9565/aa955a/meta.

[42] Pang, T., Zheng, H., Quan, Y. & Ji, H. Recorrupted-to-recorrupted: Unsupervised deep learning for image denoising. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2043–2052 (2021). URL https://openaccess.thecvf.com/content/CVPR2021/html/Pang_Recorrupted-to-Rec

orrupted_Unsupervised_Deep_Learning_for_Image_Denoising_CVPR_2021_paper.html.

[43] Sutton, R. S. & Barto, A. G. *Reinforcement learning: an introduction* (MIT press, 2018), second edition edn.

[44] Silver, D. *et al.* Mastering the game of go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016). URL https://doi.org/10.1038/nature16961.

[45] Silver, D. *et al.* Mastering the game of go without human knowledge. *Nature* **550**, 354–359 (2017). URL https://doi.org/10.1038/nature24270.

[46] Andreasson, P., Johansson, J., Liljestrand, S. & Granath, M. Quantum error correction for the toric code using deep reinforcement learning. *Quantum* **3**, 183 (2019). URL https://doi.org/10.22331%2Fq-2019-09-02-183.

[47] Graesser, L. & Keng, W. L. *Foundations of Deep Reinforcement Learning - Theory and Practice in Python.* Addison Wesley Data and Analytics Series (Pearson Education, Inc., 2020), 1st edn.

[48] Seif, A. *et al.* Machine learning assisted readout of trapped-ion qubits. *Journal of Physics B: Atomic, Molecular and Optical Physics* **51**, 174006 (2018). URL https://iopscience.iop.org/article/10.1088/1361-6455/aad62b.

[49] Melnikov, A. A. *et al.* Active learning machine learns to create new quantum experiments. *Proceedings of the National Academy of Sciences* **115**, 1221–1226 (2018). URL https://doi.org/10.1073/pnas.1714936115.

[50] Kalantre, S. S. *et al.* Machine learning techniques for state recognition and auto-tuning in quantum dots. *npj Quantum Information* **5** (2019). URL https://doi.org/10.1038/s41534-018-0118-7.

[51] Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information* (Cambridge University Press, 2010), 10th anniversary edition edn. URL https://doi.org/10.1017/CBO9780511976667.

[52] Feynman, R. P. Simulating physics with computers. *International Journal of Theoretical Physics* **21**, 467–488 (1982). URL https://doi.org/10.1007/BF02650179.

[53] Deutsch, D. & Jozsa, R. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* **439**, 553–558 (1992). URL https://doi.org/10.1098/rspa.1992.0167.

[54] Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, STOC '96, 212–219 (ACM Press, 1996). URL https://doi.org/10.1145/237814.237866.

[55] Shor, P. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, SFCS-94 (IEEE Comput. Soc. Press, 1994). URL https://ieeexplore.ieee.org/document/365700.

[56] Schaller, R. Moore's law: past, present and future. *IEEE Spectrum* **34**, 52–59 (1997). URL https://doi.org/10.1109/6.591665.

[57] Prati, E., Rotta, D., Sebastiano, F. & Charbon, E. From the quantum moore's law toward silicon based universal quantum computing. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 1–4 (2017). URL https://ieeexplore.ieee.org/document/8123662.

[58] Madsen, L. S. *et al.* Quantum computational advantage with a programmable photonic processor. *Nature* **606**, 75–81 (2022). URL https://doi.org/10.1038/s41586-022-04725-x.

[59] Egan, L. *et al.* Fault-tolerant operation of a quantum error-correction code (2021). URL https://arxiv.org/abs/2009.11482. 2009.11482.

[60] Wurtz, J. *et al.* Aquila: Quera's 256-qubit neutral-atom quantum computer (2023). URL https://arxiv.org/abs/2306.11727. 2306.11727.

[61] Norcia, M. A. *et al.* Iterative assembly of ^{171}yb atom arrays with cavity-enhanced optical lattices. *PRX Quantum* **5**, 030316 (2024). URL https://link.aps.org/doi/10.1103/PRXQuantum.5.030316.

[62] Barnes, K. *et al.* Assembly and coherent control of a register of nuclear spin qubits. *Nature Communications* **13** (2022). URL https://www.nature.com/articles/s41467-022-29977-z.

[63] Pelegrí, G., Daley, A. J. & Pritchard, J. D. High-fidelity multiqubit rydberg gates via two-photon adiabatic rapid passage. *Quantum Science and Technology* **7**, 045020 (2022). URL http://dx.doi.org/10.1088/2058-9565/ac823a.

[64] Wang, D., Sundaram, A., Kothari, R., Kapoor, A. & Roetteler, M. Quantum algorithms for reinforcement learning with a generative model (2021). URL https://arxiv.org/abs/2112.08451. 2112.08451.

[65] Aïmeur, E., Brassard, G. & Gambs, S. Quantum speed-up for unsupervised learning. *Mach. Learn.* **90**, 261–287 (2013). URL https://doi.org/10.1007/s10994-012-5316-5.

[66] Zeguendry, A., Jarir, Z. & Quafafou, M. Quantum machine learning: A review and case studies. *Entropy* **25**, 287 (2023). URL https://doi.org/10.3390/e25020287.

[67] Aïmeur, E., Brassard, G. & Gambs, S. Machine learning in a quantum world. In Lamontagne, L. & Marchand, M. (eds.) *Advances in Artificial Intelligence*, 431–442 (Springer Berlin Heidelberg, 2006). URL https://link.springer.com/chapter/10.1007/11766247_37.

[68] Alchieri, L., Badalotti, D., Bonardi, P. & Bianco, S. An introduction to quantum machine learning: from quantum logic to quantum deep learning. *Quantum Mach. Intell.* **3** (2021). URL https://doi.org/10.1007/s42484-021-00056-8.

[69] Stoudenmire, E. M. & Schwab, D. J. Supervised learning with quantum-inspired tensor networks (2017). URL https://arxiv.org/abs/1605.05775. 1605.05775.

[70] Paparo, G. D., Dunjko, V., Makmal, A., Martin-Delgado, M. A. & Briegel, H. J. Quantum Speedup for Active Learning Agents. *Phys. Rev. X* **4** (2014). URL https://doi.org/10.1103/PhysRevX.4.031002.

[71] Schuld, M., Sinayskiy, I. & Petruccione, F. The quest for a Quantum Neural Network. *Quantum Inf. Process.* **13**, 2567–2586 (2014). URL https://doi.org/10.1007/s11128-014-0809-8.

[72] Wiebe, N., Kapoor, A. & Svore, K. M. Quantum perceptron models. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, 4006–4014 (Curran Associates Inc., 2016). URL https://papers.nips.cc/paper_files/paper/2016/hash/d47268e9db2e9aa3827bba3afb7ff94a-Abstract.html.

[73] Cao, C., Zhang, C., Wu, Z., Grassl, M. & Zeng, B. Quantum variational learning for quantum error-correcting codes. *Quantum* **6**, 828 (2022). URL https://quantum-journal.org/papers/q-2022-10-06-828/.

[74] Johnson, P. D., Romero, J., Olson, J., Cao, Y. & Aspuru-Guzik, A. Qvector: An algorithm for device-tailored quantum error correction (2017). URL https://arxiv.org/abs/1711.02249. 1711.02249.

[75] Nautrup, H. P., Delfosse, N., Dunjko, V., Briegel, H. J. & Friis, N. Optimizing quantum error correction codes with reinforcement learning. *Quantum* **3**, 215 (2019). URL https://doi.org/10.22331%2Fq-2019-12-16-215.

[76] Sivak, V. V. *et al.* Real-time quantum error correction beyond break-even (2022). URL https://arxiv.org/abs/2211.09116. 2211.09116.

[77] Carleo, G. & Troyer, M. Solving the quantum many-body problem with artificial neural networks. *Science* **355**, 602–606 (2017). URL https://www.science.org/doi/10.1126/science.aag2302.

[78] Chen, S., Cotler, J., Huang, H.-Y. & Li, J. Exponential separations between learning with and without quantum memory (2021). URL https://arxiv.org/abs/2111.05881. 2111.05881.

[79] Huang, H.-Y. *et al.* Quantum advantage in learning from experiments. *Science* **376**, 1182–1186 (2022). URL http://dx.doi.org/10.1126/science.abn7293.

[80] Chen, S., Cotler, J., Huang, H.-Y. & Li, J. A hierarchy for replica quantum advantage (2021). URL https://arxiv.org/abs/2111.05874. 2111.05874.

[81] Cotler, J., Huang, H.-Y. & McClean, J. R. Revisiting dequantization and quantum advantage in learning tasks (2021). URL https://arxiv.org/abs/2112.00811. 2112.00811.

[82] Beer, K. *et al.* Training deep quantum neural networks. *Nature Communications* **11** (2020). URL https://doi.org/10.1038/s41467-020-14454-2.

[83] Lindner, A. & Strauch, D. *A Complete Course on Theoretical Physics: From Classical Mechanics to Advanced Quantum Statistics* (Springer International Publishing, 2018). URL https://doi.org/10.1007/978-3-030-04360-5.

[84] Heggie, D. & Hut, P. *The Gravitational Million–Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics* (Cambridge University Press, 2003). URL https://doi.org/10.1017/CBO9781139164535.

[85] Beutler, G. *Methods of Celestial Mechanics* (Springer Berlin, Heidelberg, 2005). URL https://doi.org/10.1007/b138225.

[86] Dietterich, T. G. *Machine Learning for Sequential Data: A Review*, 15–30 (Springer Berlin Heidelberg, 2002). URL https://doi.org/10.1007/3-540-70659-3_2.

[87] Lyon, C. & Frank, R. Using single layer networks for discrete, sequential data: An example from natural language processing. *Neural Computing and Applications* **5**, 196–214 (1997). URL https://doi.org/10.1007/BF01424225.

[88] Pandey, A. K. & Roy, S. S. Natural language generation using sequential models: A survey. *Neural Processing Letters* **55**, 7709–7742 (2023). URL https://doi.org/10.1007/s11063-023-11281-6.

[89] Berner, J., Grohs, P., Kutyniok, G. & Petersen, P. *The Modern Mathematics of Deep Learning*, chap. 1, 1–111 (Cambridge University Press, 2022). URL http://dx.doi.org/10.1017/9781009025096.002. https://arxiv.org/abs/2105.04026.

[90] Kretschmann, D. & Werner, R. F. Quantum channels with memory. *Phys. Rev. A* **72**, 062323 (2005). URL https://link.aps.org/doi/10.1103/PhysRevA.72.062323.

[91] Grumbling, E. & Horowitz, M. (eds.) *Quantum Computing: Progress and Prospects* (The National Academies Press, Washington, DC, 2019). URL https://nap.nationalacademies.org/catalog/25196/quantum-computing-progress-and-prospects.

[92] Larsen, M. V., Guo, X., Breum, C. R., Neergaard-Nielsen, J. S. & Andersen, U. L. Deterministic generation of a two-dimensional cluster state. *Science* **366**, 369–372 (2019). URL https://www.science.org/doi/full/10.1126/science.aay4354.

[93] Xavier, G. B. & Lima, G. Quantum information processing with space-division multiplexing optical fibres. *Communications Physics* **3**, 9 (2020). URL https://www.nature.com/articles/s42005-019-0269-7.

[94] Kim, T., Maunz, P. & Kim, J. Efficient collection of single photons emitted from a trapped ion into a single-mode fiber for scalable quantum-information processing. *Phys. Rev. A* **84**, 063423 (2011). URL https://link.aps.org/doi/10.1103/PhysRevA.84.063423.

[95] OpenAI *et al.* Solving rubik's cube with a robot hand (2019). URL https://arxiv.org/abs/1910.07113. arXiv:1910.07113.

[96] Ranaweera, M. & Mahmoud, Q. H. Bridging the reality gap between virtual and physical environments through reinforcement learning. *IEEE Access* **11**, 19914–19927 (2023). URL https://ieeexplore.ieee.org/abstract/document/10054009.

[97] Dulac-Arnold, G., Mankowitz, D. & Hester, T. Challenges of real-world reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning*, vol. 97 (PMLR, 2019). URL https://arxiv.org/abs/1904.12901.

[98] Dulac-Arnold, G. *et al.* *Challenges of real-world reinforcement learning: Definitions, benchmarks and analysis*, vol. 110, 2419–2468 (Springer, 2021). URL https://link.springer.com/article/10.1007/s10994-021-05961-4.

[99] Ding, Z. & Dong, H. *Challenges of Reinforcement Learning*, 249–272 (Springer, 2020). URL https://doi.org/10.1007/978-981-15-4095-0_7.

[100] Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. & Krause, A. (eds.) *Proceedings of the 35th International Conference on Machine Learning*, vol. 80 of *Proceedings of Machine Learning Research*, 1861–1870 (PMLR, 2018). URL https://proceedings.mlr.press/v80/haarnoja18b.html.

[101] Haarnoja, T. *et al.* Soft actor-critic algorithms and applications (2019). URL https://arxiv.org/abs/1812.05905. 1812.05905.

[102] Kuznetsov, A., Shvechikov, P., Grishin, A. & Vetrov, D. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *Proceedings of the 37th International Conference on MachineLearning*, vol. 119 (PMLR, 2020). URL https://proceedings.mlr.press/v119/kuznetsov20a/kuznetsov20a.pdf.

[103] Fujimoto, S., van Hoof, H. & Meger, D. Addressing function approximation error in actor-critic methods (2018). URL https://arxiv.org/abs/1802.09477. 1802.09477.

[104] Bondarenko, D., Salzmann, R. & Schmiesing, V.-S. Learning quantum processes with memory – quantum recurrent neural networks (2023). URL https://arxiv.org/abs/2301.08167. 2301.08167.

[105] Richtmann, L. *et al.* Model-free reinforcement learning with noisy actions for automated experimental control in optics (2024). URL https://arxiv.org/abs/2405.15421. 2405.15421.

[106] Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* **29**, 141–142 (2012). URL https://ieeexplore.ieee.org/document/6296535.

[107] Nielsen, M. *Neural Networks and Deep Learning* (Determination Press, 2015). URL http://neuralnetworksanddeeplearning.com/.

[108] Wolf, M. M. Mathematical foundations of supervised learning. Lecture Notes (2023). URL https://mediatum.ub.tum.de/doc/1723378/1723378.pdf.

[109] MacKay, D. J. C. *Information Theory, Inference, and Learning Algorithms* (Cambridge University Press, 2005), 7.2 edn.

[110] Willmott, C. J. & Matsuura, K. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate Research* **30**, 79–82 (2005). URL https://www.int-res.com/abstracts/cr/v30/n1/p79-82/.

[111] Huber, P. J. Robust estimation of a location parameter. *The Annals of Mathematical Statistics* **35**, 73–101 (1964). URL https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-35/issue-1/Robust-Estimation-of-a-Location-Parameter/10.1214/aoms/1177703732.full.

[112] Hastie, T., Tibshirani, R. & Friedman, J. *The Elements of Statistical Learning* (Springer New York, 2009). URL https://link.springer.com/book/10.1007/978-0-387-84858-7.

[113] McCulloch, W. S. & Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* **5**, 115–133 (1943). URL https://doi.org/10.1007/BF02478259.

[114] Haykin, S. *Neural Networks - A Comprehensive Foundation* (Pearson Education, Inc., 1999).

[115] LeCun, Y., Bottou, L., Bengio, Y. & Haner, P. Gradient based learning applied to document recognition. In *Proc. of the IEEE* (1998). URL https://ieeexplore.ieee.org/document/726791.

[116] He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778 (2016). URL https://ieeexplore.ieee.org/document/7780459.

[117] Young, T., Hazarika, D., Poria, S. & Cambria, E. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine* **13**, 55–75 (2018). URL https://ieeexplore.ieee.org/document/8416973.

[118] Mnih, V. *et al.* Playing atari with deep reinforcement learning (2013). URL https://arxiv.org/abs/1312.5602. 1312.5602.

[119] OpenAI *et al.* Dota 2 with large scale deep reinforcement learning (2019). URL https://arxiv.org/abs/1912.06680. 1912.06680.

[120] Vinyals, O. *et al.* Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature* **575**, 350–354 (2019). URL https://doi.org/10.1038/s41586-019-1724-z.

[121] Senior, A. W. *et al.* Improved protein structure prediction using potentials from deep learning. *Nature* **577**, 706–710 (2020). URL https://doi.org/10.1038/s41586-019-1923-7.

[122] Iyyer, M., Enns, P., Boyd-Graber, J. & Resnik, P. Political ideology detection using recursive neural networks. In Toutanova, K. & Wu, H. (eds.) *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1113–1122 (Association for Computational Linguistics, 2014). URL https://aclanthology.org/P14-1105.

[123] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* **65**, 386–408 (1958). URL https://psycnet.apa.org/doiLanding?doi=10.1037/h0042519.

[124] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems* **2**, 303–314 (1989). URL https://doi.org/10.1007/BF02551274.

[125] Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4**, 251–257 (1991). URL https://www.sciencedirect.com/science/article/pii/089360809190009T.

[126] Funahashi, K.-I. On the approximate realization of continuous mappings by neural networks. *Neural Networks* **2**, 183–192 (1989). URL https://www.sciencedirect.com/science/article/pii/0893608089900038.

[127] Hornik, K., Stinchcombe, M. & White, H. Multilayer feedforward networks are universal approximators. *Neural Networks* **2**, 359–366 (1989). URL https://www.sciencedirect.com/science/article/pii/0893608089900208.

[128] Lu, Z., Pu, H., Wang, F., Hu, Z. & Wang, L. The expressive power of neural networks: A view from the width. In Guyon, I. *et al.* (eds.) *Advances in Neural Information Processing Systems*, vol. 30 (Curran Associates, Inc., 2017). URL https://proceedings.neurips.cc/paper_files/paper/2017/file/32cbf687880eb1674a07bf717761dd3a-Paper.pdf.

[129] Kratsios, A. & Papon, L. Universal approximation theorems for differentiable geometric deep learning. *Journal of Machine Learning Research* **23**, 1–73 (2022). URL http://jmlr.org/papers/v23/21-0716.html.

[130] Kidger, P. & Lyons, T. Universal approximation with deep narrow networks. In Abernethy, J. & Agarwal, S. (eds.) *Proceedings of Thirty Third Conference on Learning Theory*, vol. 125 of *Proceedings of Machine Learning Research*, 2306–2327 (PMLR, 2020). URL https://proceedings.mlr.press/v125/kidger20a.html.

[131] Maiorov, V. & Pinkus, A. Lower bounds for approximation by mlp neural networks. *Neurocomputing* **25**, 81–91 (1999). URL https://www.sciencedirect.com/science/article/pii/S0925231298001118.

[132] Guliyev, N. J. & Ismailov, V. E. Approximation capability of two hidden layer feedforward neural networks with fixed weights. *Neurocomputing* **316**, 262–269 (2018). URL https://www.sciencedirect.com/science/article/pii/S0925231218309111.

[133] Boyd, S. & Vandenberghe, L. *Convex Optimization* (Cambridge University Press, Cambridge, 2004). URL https://www.cambridge.org/core/product/17D2FAA54F641A2F62C7CCD01DFA97C4.

[134] Werbos, P. *Beyond Regression : New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University (1974).

[135] Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *Nature* **323**, 533–536 (1986). URL https://doi.org/10.1038/323533a0.

[136] Medsker, L. R. & Jain, L. C. (eds.). *Recurrent Neural Networks. Design and Applications*, chap. Introduction, 13–24 (CRC Press LLC, 2001). URL https://doi.org/10.1201/9781003040620.

[137] Mikolov, T., Karafiát, M., Burget, L., Cernocký, J. H. & Khudanpur, S. Recurrent neural network based language model. In *Interspeech* (2010). URL https://www.isca-archive.org/interspeech_2010/mikolov10_interspeech.html#.

[138] Costa, M., Pasero, E., Piglione, F. & Radasanu, D. Short term load forecasting using a synchronously operated recurrent neural network. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, vol. 5, 3478–3482 vol.5 (1999). URL https://ieeexplore.ieee.org/abstract/document/836225.

[139] Kalchbrenner, N. & Blunsom, P. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1700–1709 (Association for Computational Linguistics, 2013). URL https://aclanthology.org/D13-1176.

[140] Yin, W., Kann, K., Yu, M. & Schütze, H. Comparative study of cnn and rnn for natural language processing (2017). URL https://arxiv.org/abs/1702.01923. 1702.01923.

[141] Chen, G. A gentle tutorial of recurrent neural network with error backpropagation (2018). URL https://arxiv.org/abs/1610.02583. 1610.02583.

[142] Werbos, P. J. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks* **1**, 339–356 (1988). URL https://www.sciencedirect.com/science/article/pii/089360808890007X.

[143] Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9**, 1735–80 (1997). URL https://direct.mit.edu/neco/article-abstract/9/8/1735/6109/Long-Short-Term-Memory?redirectedFrom=fulltext.

[144] Chung, J., Gulcehre, C., Cho, K. H. & Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling (2014). URL https://arxiv.org/abs/1412.3555. 1412.3555.

[145] Mohri, M., Rostamizadeh, A. & Talkwalkar, A. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning Series (MIT Press, 2018), 2nd edn. URL https://cs.nyu.edu/~mohri/mlbook/.

[146] Ripley, B. D. *Pattern Recognition and Neural Networks* (Cambridge University Press, 1996). URL https://www.cambridge.org/core/product/4E038249C9BAA06C8F4EE6F044D09C5C.

[147] Silver, D. *et al.* A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* **362**, 1140–1144 (2018). URL https://doi.org/10.1126/science.aar6404.

[148] Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015). URL https://doi.org/10.1038/nature14236.

[149] Fawzi, A. *et al.* Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* **610**, 47–53 (2022). URL https://doi.org/10.1038/s41586-022-05172-4.

[150] Ruiz, F. J. R. *et al.* Quantum circuit optimization with alphatensor (2024). URL https://arxiv.org/abs/2402.14396. 2402.14396.

[151] Silver, D. URL https://www.davidsilver.uk/teaching/.

[152] OpenAI. URL https://spinningup.openai.com/en/latest/index.html.

[153] Begain, K. *et al.* Theoretical background. In *Practical Performance Modeling: Application of the MOSEL Language*, The Springer International Series in Engineering and Computer Science, chap. 2, 9–61 (Springer US, Boston, MA, 2001). URL https://doi.org/10.1007/978-1-4615-1387-2_2.

[154] Spaan, M. T. J. Partially observable markov decision processes. In Wiering, M. & van Otterlo, M. (eds.) *Reinforcement Learning: State-of-the-Art* (Springer-Verlag Berlin Heidelberg, 2012). URL https://www.st.ewi.tudelft.nl/mtjspaan/pub/Spaan12pomdp.pdf.

[155] Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. Trust region policy optimization. In Bach, F. & Blei, D. (eds.) *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37 of *Proceedings of Machine Learning Research*, 1889–1897 (PMLR, 2015). URL https://proceedings.mlr.press/v37/schulman15.html.

[156] Schaul, T., Quan, J., Antonoglou, I. & Silver, D. Prioritized experience replay (2016). URL https://arxiv.org/abs/1511.05952. 1511.05952.

[157] Andrychowicz, M. *et al.* Hindsight experience replay. In Guyon, I. *et al.* (eds.) *Advances in Neural Information Processing Systems*, vol. 30 (Curran Associates, Inc., 2017). URL https://proceedings.neurips.cc/paper_files/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf.

[158] van Hasselt, H., Guez, A. & Silver, D. Deep reinforcement learning with double q-learning (2015). URL https://arxiv.org/abs/1509.06461. 1509.06461.

[159] Wang, Z. *et al.* Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning - Volume 48*, 1995–2003 (JMLR.org, 2016). URL https://proceedings.mlr.press/v48/wangf16.html.

[160] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* **8**, 229–256 (1992). URL https://doi.org/10.1007/BF00992696.

[161] Williams, R. & Peng, J. Function optimization using connectionist reinforcement learning algorithms. *Connection Science* **3**, 241–268 (1991). URL https://doi.org/10.1080/09540099108946587.

[162] Mnih, V. *et al.* Asynchronous methods for deep reinforcement learning. In Balcan, M. F. & Weinberger, K. Q. (eds.) *Proceedings of The 33rd International Conference on Machine Learning*, vol. 48 of *Proceedings of Machine Learning Research*, 1928–1937 (PMLR, 2016). URL https://proceedings.mlr.press/v48/mniha16.html.

[163] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal policy optimization algorithms (2017). URL https://arxiv.org/abs/1707.06347. 1707.06347.

[164] Lillicrap, T. P. *et al.* Continuous control with deep reinforcement learning (2019). URL https://arxiv.org/abs/1509.02971. 1509.02971.

[165] Dabney, W., Rowland, M., Bellemare, M. & Munos, R. Distributional reinforcement learning with quantile regression. *Proceedings of the AAAI Conference on Artificial Intelligence* **32** (2018). URL https://ojs.aaai.org/index.php/AAAI/article/view/11791.

[166] Hayashi, M. *Quantum Information Theory* (Springer-Verlag Berlin Heidelberg, 2017), 2 edn. URL https://doi.org/10.1007/978-3-662-49725-8.

[167] Heinosaari, T. & Ziman, M. *The Mathematical Language of Quantum Theory: From Uncertainty to Entanglement* (Cambridge University Press, 2011). URL https://www.cambridge.org/core/product/D8AAEF727B99D7AB098F9162C6D55FC8.

[168] Kreyszig, E. *Introductory Functional Analysis with Applications* (Wiley and Sons Ltd, 1989).

[169] Dirac, P. A. M. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society* **35**, 416–418 (1939). URL https://www.cambridge.org/core/product/4631DB9213D680D6332BA11799D76AFB.

[170] Reed, M. & Simon, B. *Functional analysis*, vol. 1 of *Methods of Modern Mathematical Physics* (Academic Press, San Diego, Calif. [u.a.], 1980).

[171] Halmos, P. R. *A Hilbert Space Problem Book* (Springer New York, 1982). URL https://link.springer.com/book/10.1007/978-1-4684-9330-6.

[172] Liu, S. & Trenkler, G. Hadamard, khatri-rao, kronecker and other matrix products. *International Journal of Information and Systems Sciences* **4** (2008). URL https://www.researchgate.net/publication/251677036_Hadamard_Khatri-Rao_Kronecker_and_other_matrix_products.

[173] Rybár, T. & Ziman, M. Quantum finite-depth memory channels: Case study. *Phys. Rev. A* **80**, 042306 (2009). URL https://link.aps.org/doi/10.1103/PhysRevA.80.042306.

[174] Rybár, T. & Ziman, M. Process estimation in the presence of time-invariant memory effects. *Phys. Rev. A* **92**, 042315 (2015). URL https://link.aps.org/doi/10.1103/PhysRevA.92.042315.

[175] Ball, K., Carlen, E. A. & Lieb, E. H. Sharp uniform convexity and smoothness inequalities for trace norms. *Inventiones mathematicae* **115**, 463–482 (1994). URL https://doi.org/10.1007/BF01231769.

[176] Jozsa, R. Fidelity for mixed quantum states. *Journal of Modern Optics* **41**, 2315–2323 (1994). URL https://doi.org/10.1080/09500349414552171.

[177] Frankel, T. *The Geometry of Physics, an Introduction* (Cambridge University Press, 2003), second edition edn. URL https://doi.org/10.1017/CBO9781139061377.

[178] Halmos, P. R. *Measure Theory*, vol. 18 of *Graduate Texts in Mathematics* (Springer-Verlag New York, 1950). URL https://doi.org/10.1007/978-1-4684-9440-2.

[179] Życzkowski, K. & Sommers, H.-J. Induced measures in the space of mixed quantum states. *Journal of Physics A: Mathematical and General* **34**, 7111–7125 (2001). URL https://iopscience.iop.org/article/10.1088/0305-4470/34/35/335/pdf.

[180] Wootters, W. K. Random quantum states. *Foundations of Physics* **20** (1990). URL https://link.springer.com/content/pdf/10.1007/BF01883491.pdf.

[181] Hayden, P., Leung, D. W. & Winter, A. Aspects of generic entanglement. *Communications in Mathematical Physics* **265**, 95–117 (2006). URL https://link.springer.com/content/pdf/10.1007/s00220-006-1535-6.pdf.

[182] Bridgeman, J. C. & Chubb, C. T. Hand-waving and interpretive dance: an introductory course on tensor networks. *Journal of Physics A: Mathematical and Theoretical* **50**, 223001 (2017). URL http://dx.doi.org/10.1088/1751-8121/aa6dc3.

[183] Honeywell. *Quantinuum System Model H2 Product Data Sheet*, 1.00 edn. (2023). URL https://assets.website-files.com/62b9d45fb3f64842a96c9686/6459acc9b999bb7fb526c4bf_QuantinuumH2ProductDataSheet.pdf.

[184] IBM. URL https://www.ibm.com/quantum/blog/quantum-roadmap-2033.

[185] Rigetti. Ankaa-2 quantum processor. URL https://qcs.rigetti.com/qpus.

[186] Xinhua. China launches 504-qubit quantum chip, open to global users. China Daily (2024). URL https://www.chinadaily.com.cn/a/202404/26/WS662b15dfa31082fc043c431e.html.

[187] Rigetti. Rigetti announces public availability of ankaa-2 system with a 2.5x performance improvement compared to previous qpus. GlobalNewswire (2024). URL https://www.globenewswire.com/news-release/2024/01/04/2804006/0/en/Rigetti-Announces-Public-Availability-of-Ankaa-2-System-with-a-2-5x-Performance-Improvement-Compared-to-Previous-QPUs.html.

[188] IQM. Finland launches a 20-qubit quantum computer – development towards more powerful quantum computers continues (2023). URL https://www.meetiqm.com/newsroom/press-releases/finland-launches-a-20-qubit-quantum-computer.

[189] M Squared Lasers. *Maxwell: Neutral Atom Quantum Processor*. URL https://m2lasers.com/quantum-datasheet.html?file=Maxwell_Explainer.pdf.

[190] Wilkins, A. Record-breaking quantum computer has more than 1000 qubits. NewScientist (2023). URL https://www.newscientist.com/article/2399246-record-breaking-quantum-computer-has-more-than-1000-qubits/.

[191] Padavic-Callaghan, K. Ibm's 'condor' quantum computer has more than 1000 qubits. New Scientist (2023). URL https://www.newscientist.com/article/2405789-ibms-condor-quantum-computer-has-more-than-1000-qubits/.

[192] Preskill, J. Quantum computing in the nisq era and beyond. *Quantum* **2**, 79 (2018). URL http://dx.doi.org/10.22331/q-2018-08-06-79.

[193] Lovett, N. B., Crosnier, C., Perarnau-Llobet, M. & Sanders, B. C. Differential Evolution for Many-Particle Adaptive Quantum Metrology. *Phys. Rev. Lett.* **110** (2013). URL https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.110.220501.

[194] Tiersch, M., Ganahl, E. J. & Briegel, H. J. Adaptive quantum computation in changing environments using projective simulation. *Sci. Rep.* **5**, 12874 (2015). URL https://www.nature.com/articles/srep12874.

[195] Torlai, G. & Melko, R. G. Neural decoder for topological codes. *Physical Review Letters* **119**, 030501 (2017). URL https://doi.org/10.1103/PhysRevLett.119.030501.

[196] Krastanov, S. & Jiang, L. Deep neural network probabilistic decoder for stabilizer codes. *Scientific Reports* **7**, 11003 (2017). URL https://doi.org/10.1038/s41598-017-11266-1.

[197] Maskara, N., Kubica, A. & Jochym-O'Connor, T. Advantages of versatile neural-network decoding for topological codes. *Phys. Rev. A* **99**, 052351 (2019). URL https://link.aps.org/doi/10.1103/PhysRevA.99.052351.

[198] Baireuther, P., O'Brien, T. E., Tarasinski, B. & Beenakker, C. W. J. Machine-learning-assisted correction of correlated qubit errors in a topological code. *Quantum* **2**, 48 (2018). URL https://doi.org/10.22331/q-2018-01-29-48.

[199] Varsamopoulos, S., Bertels, K. & Almudever, C. G. Decoding surface code with a distributed neural network based decoder (2019). URL https://arxiv.org/abs/1901.10847. 1901.10847.

[200] Chamberland, C. & Ronagh, P. Deep neural decoders for near term fault-tolerant experiments. *Quantum Science and Technology* **3**, 044002 (2018). URL https://iopscience.iop.org/article/10.1088/2058-9565/aad1f7/meta.

[201] Chen, H., Vasmer, M., Breuckmann, N. P. & Grant, E. Automated discovery of logical gates for quantum error correction. *Quantum Information and Computation* **22**, 947–964 (2022). URL https://doi.org/10.26421%2Fqic22.11-12-3.

[202] Fösel, T., Tighineanu, P., Weiss, T. & Marquardt, F. Reinforcement learning with neural networks for quantum feedback. *Phys. Rev. X* **8**, 031084 (2018). URL https://link.aps.org/doi/10.1103/PhysRevX.8.031084.

[203] Sweke, R., Kesselring, M. S., van Nieuwenburg, E. P. L. & Eisert, J. Reinforcement learning decoders for fault-tolerant quantum computation. *Machine Learning: Science and Technology* **2**, 025005 (2020). URL https://iopscience.iop.org/article/10.1088/2632-2153/abc609.

[204] Gebhart, V. *et al.* Learning quantum systems. *Nature Reviews Physics* **5**, 141–156 (2023). URL https://doi.org/10.1038/s42254-022-00552-1.

[205] Peral-García, D., Cruz-Benito, J. & García-Peñalvo, F. J. Systematic literature review: Quantum machine learning and its applications. *Computer Science Review* **51**, 100619 (2024). URL https://doi.org/10.1016/j.cosrev.2024.100619.

[206] Skolik, A., Jerbi, S. & Dunjko, V. Quantum agents in the gym: a variational quantum algorithm for deep q-learning. *Quantum* **6**, 720 (2022). URL http://dx.doi.org/10.22331/q-2022-05-24-720.

[207] Schuld, M., Bocharov, A., Svore, K. M. & Wiebe, N. Circuit-centric quantum classifiers. *Phys. Rev. A* **101**, 032308 (2020). URL https://link.aps.org/doi/10.1103/PhysRevA.101.032308.

[208] Henderson, M., Shakya, S., Pradhan, S. & Cook, T. Quanvolutional neural networks: Powering image recognition with quantum circuits (2019). URL https://arxiv.org/abs/1904.04767. 1904.04767.

[209] Pérez-Salinas, A., Cervera-Lierta, A., Gil-Fuster, E. & Latorre, J. I. Data re-uploading for a universal quantum classifier. *Quantum* **4**, 226 (2020). URL https://doi.org/10.22331/q-2020-02-06-226.

[210] Wei, S., Chen, Y., Zhou, Z. & Long, G. A quantum convolutional neural network on nisq devices. *AAPPS Bulletin* **32** (2022). URL https://doi.org/10.1007/s43673-021-00030-3.

[211] Zhang, K., Hsieh, M.-H., Liu, L. & Tao, D. Toward trainability of quantum neural networks (2020). URL https://arxiv.org/abs/2011.06258. 2011.06258.

[212] Mari, A., Bromley, T. R., Izaac, J., Schuld, M. & Killoran, N. Transfer learning in hybrid classical-quantum neural networks. *Quantum* **4**, 340 (2020). URL http://dx.doi.org/10.22331/q-2020-10-09-340.

[213] Havlíček, V. *et al.* Supervised learning with quantum-enhanced feature spaces. *Nature* **567**, 209–212 (2019). URL https://doi.org/10.1038/s41586-019-0980-2.

[214] Zoufal, C., Lucchi, A. & Woerner, S. Variational quantum boltzmann machines. *Quantum Machine Intelligence* **3** (2021). URL http://dx.doi.org/10.1007/s42484-020-00033-7.

[215] Huang, H.-Y. *et al.* Power of data in quantum machine learning. *Nature Communications* **12** (2021). URL https://doi.org/10.1038/s41467-021-22539-9.

[216] Wilson, C. M. *et al.* Quantum kitchen sinks: An algorithm for machine learning on near-term quantum computers (2019). URL https://arxiv.org/abs/1806.08321. 1806.08321.

[217] Lloyd, S., Schuld, M., Ijaz, A., Izaac, J. & Killoran, N. Quantum embeddings for machine learning (2020). URL https://arxiv.org/abs/2001.03622. 2001.03622.

[218] Bowles, J., Ahmed, S. & Schuld, M. Better than classical? the subtle art of benchmarking quantum machine learning models (2024). URL https://arxiv.org/abs/2403.07059. 2403.07059.

[219] Wang, D., You, X., Li, T. & Childs, A. M. Quantum exploration algorithms for multi-armed bandits. *Proceedings of the AAAI Conference on Artificial Intelligence* **35**, 10102–10110 (2021). URL https://doi.org/10.1609/aaai.v35i11.17212.

[220] Schuld, M., Sinayskiy, I. & Petruccione, F. An introduction to quantum machine learning. *Contemporary Physics* **56**, 172–185 (2014). URL https://doi.org/10.1080/00107514.2014.964942.

[221] Dunjko, V. & Briegel, H. J. Machine learning & artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics* **81**, 074001 (2018). URL https://doi.org/10.1088/1361-6633/aab406.

[222] Sasaki, M. & Carlini, A. Quantum learning and universal quantum matching machine. *Phys. Rev. A* **66**, 022303 (2002). URL https://link.aps.org/doi/10.1103/PhysRevA.66.022303.

[223] Gambs, S. Quantum classification (2008). URL https://arxiv.org/abs/0809.0444.

[224] Sentís, G., Calsamiglia, J., Muñoz-Tapia, R. & Bagan, E. Quantum learning without quantum memory. *Sci. Rep.* **2**, 708 (2012). URL https://doi.org/10.1038/srep00708.

[225] Dunjko, V., Taylor, J. M. & Briegel, H. J. Quantum-enhanced machine learning. *Phys. Rev. Lett.* **117**, 130501 (2016). URL https://link.aps.org/doi/10.1103/PhysRevLett.117.130501.

[226] Monràs, A., Sentís, G. & Wittek, P. Inductive supervised quantum learning. *Phys. Rev. Lett.* **118**, 190503 (2017). URL https://link.aps.org/doi/10.1103/PhysRevLett.118.190503.

[227] Alvarez-Rodriguez, U., Lamata, L., Escandell-Montero, P., Martín-Guerrero, J. D. & Solano, E. Supervised quantum learning without measurements. *Sci. Rep.* **7**, 13645 (2017). URL https://doi.org/10.1038/s41598-017-13378-0.

[228] Amin, M. H., Andriyash, E., Rolfe, J., Kulchytskyy, B. & Melko, R. Quantum boltzmann machine. *Phys. Rev. X* **8**, 021050 (2018). URL https://link.aps.org/doi/10.1103/PhysRevX.8.021050.

[229] Du, Y., Hsieh, M.-H., Liu, T. & Tao, D. Expressive power of parametrized quantum circuits. *Phys. Rev. Research* **2**, 033125 (2020). URL https://link.aps.org/doi/10.1103/PhysRevResearch.2.033125.

[230] Sentís, G., Monràs, A., Muñoz-Tapia, R., Calsamiglia, J. & Bagan, E. Unsupervised classification of quantum data. *Phys. Rev. X* **9**, 041029 (2019). URL https://link.aps.org/doi/10.1103/PhysRevX.9.041029.

[231] Gyurik, C., Molteni, R. & Dunjko, V. Limitations of measure-first protocols in quantum machine learning (2023). URL https://arxiv.org/abs/2311.12618. 2311.12618.

[232] Dunjko, V., Taylor, J. M. & Briegel, H. J. Advances in quantum reinforcement learning. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (IEEE, 2017). URL https://ieeexplore.ieee.org/abstract/document/8122616.

[233] Mangini, S., Tacchino, F., Gerace, D., Bajoni, D. & Macchiavello, C. Quantum computing models for artificial neural networks. *Europhysics Letters* **134**, 10002 (2021). URL https://iopscience.iop.org/article/10.1209/0295-5075/134/10002.

[234] Farhi, E. & Neven, H. Classification with quantum neural networks on near term processors (2018). URL https://arxiv.org/abs/1802.06002. 1802.06002.

[235] Chen, H., Wossnig, L., Severini, S., Neven, H. & Mohseni, M. Universal discriminative quantum neural networks. *Quantum Machine Intelligence* **3** (2020). URL http://dx.doi.org/10.1007/s42484-020-00025-7.

[236] Huang, H.-L. *et al.* Experimental quantum generative adversarial networks for image generation. *Physical Review Applied* **16** (2021). URL http://dx.doi.org/10.1103/PhysRevApplied.16.024051.

[237] Wan, K. H., Dahlsten, O., Kristjánsson, H., Gardner, R. & Kim, M. S. Quantum generalisation of feedforward neural networks. *npj Quantum Information* **3** (2017). URL https://doi.org/10.1038/s41534-017-0032-4.

[238] Schuld, M., Sweke, R. & Meyer, J. J. Effect of data encoding on the expressive power of variational quantum-machine-learning models. *Physical Review A* **103** (2021). URL http://dx.doi.org/10.1103/PhysRevA.103.032430.

[239] Dallaire-Demers, P.-L. & Killoran, N. Quantum generative adversarial networks. *Phys. Rev. A* **98**, 012324 (2018). URL https://www.semanticscholar.org/paper/10eec18a20954f9ac4d8f463524e5f160e23895f.

[240] Lloyd, S. & Weedbrook, C. Quantum generative adversarial learning. *Phys. Rev. Lett.* **121**, 040502 (2018). URL https://link.aps.org/doi/10.1103/PhysRevLett.121.040502.

[241] Cong, I., Choi, S. & Lukin, M. D. Quantum convolutional neural networks. *Nature Physics* **15**, 1273–1278 (2019). URL https://doi.org/10.1038/s41567-019-0648-8.

[242] Grant, E. *et al.* Hierarchical quantum classifiers. *npj Quantum Information* **4** (2018). URL https://doi.org/10.1038/s41534-018-0116-9.

[243] Killoran, N. *et al.* Continuous-variable quantum neural networks. *Physical Review Research* **1**, 033063 (2019). URL https://journals.aps.org/prresearch/abstract/10.1103/PhysRevResearch.1.033063.

[244] Beer, K., List, D., Müller, G., Osborne, T. J. & Struckmann, C. Training quantum neural networks on nisq devices (2021). URL https://arxiv.org/abs/2104.06081. 2104.06081.

[245] Beer, K. *Quantum neural networks*. Ph.D. thesis, Leibniz University of Hannover (2022). URL https://www.repo.uni-hannover.de/handle/123456789/11991.

[246] Bondarenko, D. *Constructing Networks of Quantum Channels for State Preparation*. Ph.D. thesis, Leibniz Universität Hannover (2021). URL https://doi.org/10.15488/11050.

[247] Locher, D. F., Cardarelli, L. & Müller, M. Quantum error correction with quantum autoencoders. *Quantum* **7**, 942 (2023). URL https://quantum-journal.org/papers/q-2023-03-09-942/.

[248] Sharma, K., Cerezo, M., Cincio, L. & Coles, P. J. Trainability of dissipative perceptron-based quantum neural networks. *Physical Review Letters* **128**, 180505 (2022). URL http://dx.doi.org/10.1103/PhysRevLett.128.180505.

[249] Beer, K., Khosla, M., Köhler, J., Osborne, T. J. & Zhao, T. Quantum machine learning of graph-structured data. *Phys. Rev. A* **108**, 012410 (2023). URL https://link.aps.org/doi/10.1103/PhysRevA.108.012410.

[250] Beer, K., List, D., Müller, G., Osborne, T. J. & Struckmann, C. Training quantum neural networks on nisq devices (2021). URL https://arxiv.org/abs/2104.06081. 2104.06081.

[251] Beer, K. & Müller, G. Dissipative quantum generative adversarial networks (2021). URL https://arxiv.org/abs/2112.06088. 2112.06088.

[252] Renziehausen, N. G. *Quantum Neural Networks with General Output*. Bachelor's thesis, Leibniz University of Hannover (2022). URL https://itp.uni-hannover.de/fileadmin/itp/qinfo/Team_Tobias_Osborne/Bachelors_Theses/Nils_Renziehausen_Bachelors_Thesis.pdf.

[253] Chua, Z. *Learning Quantum Operations - Classical vs. QuantumDissipative Neural Networks*. Master's thesis, Leibniz University of Hannover (2024). URL https://www.itp.uni-hannover.de/fileadmin/itp/qinfo/Team_Tobias_Osborne/Masters_Theses/Zi_Chua_Masters_Thesis.pdf.

[254] Cerezo, M., Verdon, G., Huang, H.-Y., Cincio, L. & Coles, P. J. Challenges and opportunities in quantum machine learning. *Nature Computational Science* **2**, 567–576 (2022). URL https://www.nature.com/articles/s43588-022-00311-3.

[255] Cerezo, M. & Coles, P. J. Higher order derivatives of quantum neural networks with barren plateaus. *Quantum Science and Technology* **6**, 035006 (2021). URL https://iopscience.iop.org/article/10.1088/2058-9565/abf51a.

[256] Holmes, Z. *et al.* Barren plateaus preclude learning scramblers. *Phys. Rev. Lett.* **126**, 190501 (2021). URL https://link.aps.org/doi/10.1103/PhysRevLett.126.190501.

[257] Binkowski, L., Koßmann, G., Osborne, T. J., Schwonnek, R. & Ziegler, T. From barren plateaus through fertile valleys: Conic extensions of parameterised quantum circuits (2023). URL https://arxiv.org/abs/2310.04255. 2310.04255.

[258] McClean, J. R., Boixo, S., Smelyanskiy, V. N., Babbush, R. & Neven, H. Barren plateaus in quantum neural network training landscapes. *Nature Communications* **9** (2018). URL https://doi.org/10.1038/s41467-018-07090-4.

[259] Arrasmith, A., Cerezo, M., Czarnik, P., Cincio, L. & Coles, P. J. Effect of barren plateaus on gradient-free optimization. *Quantum* **5**, 558 (2021). URL https://doi.org/10.22331/q-2021-10-05-558.

[260] Holmes, Z., Sharma, K., Cerezo, M. & Coles, P. J. Connecting ansatz expressibility to gradient magnitudes and barren plateaus. *PRX Quantum* **3**, 010313 (2022). URL https://link.aps.org/doi/10.1103/PRXQuantum.3.010313.

[261] Pesah, A. *et al.* Absence of barren plateaus in quantum convolutional neural networks. *Phys. Rev. X* **11**, 041011 (2021). URL https://link.aps.org/doi/10.1103/PhysRevX.11.041011.

[262] Volkoff, T. & Coles, P. J. Large gradients via correlation in random parameterized quantum circuits. *Quantum Science and Technology* **6**, 025008 (2021). URL https://doi.org/10.1088/2058-9565/abd891.

[263] Verdon, G. *et al.* Learning to learn with quantum neural networks via classical neural networks (2019). URL https://arxiv.org/abs/1907.05415. 1907.05415.

[264] Cerezo, M., Sone, A., Volkoff, T., Cincio, L. & Coles, P. J. Cost function dependent barren plateaus in shallow parametrized quantum circuits. *Nature Communications* **12** (2021). URL http://dx.doi.org/10.1038/s41467-021-21728-w.

[265] Uvarov, A. V. & Biamonte, J. D. On barren plateaus and cost function locality in variational quantum algorithms. *Journal of Physics A: Mathematical and Theoretical* **54**, 245301 (2021). URL https://iopscience.iop.org/article/10.1088/1751-8121/abfac7/meta.

[266] Wang, S. *et al.* Noise-induced barren plateaus in variational quantum algorithms. *Nature Communications* **12**, 6961 (2021). URL https://doi.org/10.1038/s41467-021-27045-6.

[267] Ortiz Marrero, C., Kieferová, M. & Wiebe, N. Entanglement-induced barren plateaus. *PRX Quantum* **2**, 040316 (2021). URL https://link.aps.org/doi/10.1103/PRXQuantum.2.040316.

[268] Patti, T. L., Najafi, K., Gao, X. & Yelin, S. F. Entanglement devised barren plateau mitigation. *Physical Review Research* **3**, 033090 (2021). URL https://doi.org/10.1103/PhysRevResearch.3.033090.

[269] Kübler, J. M., Buchholz, S. & Schölkopf, B. The inductive bias of quantum kernels (2021). URL https://arxiv.org/abs/2106.03747. 2106.03747.

[270] Hubregtsen, T. *et al.* Training quantum embedding kernels on near-term quantum computers. *Phys. Rev. A* **106**, 042431 (2022). URL https://link.aps.org/doi/10.1103/PhysRevA.106.042431.

[271] Thanasilp, S., Wang, S., Nghiem, N. A., Coles, P. & Cerezo, M. Subtleties in the tainability of quantum machine learning models. *Quantum Machine Intelligence* **5** (2023). URL https://doi.org/10.1007/s42484-023-00103-6.

[272] Perrier, E., Youssry, A. & Ferrie, C. Qdataset: Quantum datasets for machine learning (2021). URL https://arxiv.org/abs/2108.06661. 2108.06661.

[273] Schatzki, L., Arrasmith, A., Coles, P. J. & Cerezo, M. Entangled datasets for quantum machine learning (2021). URL https://arxiv.org/abs/2109.03400. 2109.03400.

[274] Dong, D. & Petersen, I. Quantum control theory and applications: a survey. *IET Control Theory & Applications* **4**, 2651–2671(20) (2010). URL https://digital-library.theiet.org/content/journals/10.1049/iet-cta.2009.0508.

[275] D'Alessandro, D. *Introduction to Quantum Control and Dynamics*. Advances in Applied Mathematics (CRC Press, 2021), second edn. URL https://www.taylorfrancis.com/books/mono/10.1201/9781003051268/introduction-quantum-control-dynamics-domenico-alessandro.

[276] Behrman, E., Nash, L., Steck, J., Chandrashekar, V. & Skinner, S. Simulations of quantum neural networks. *Information Sciences* **128**, 257–269 (2000). URL https://www.sciencedirect.com/science/article/pii/S0020025500000566.

[277] Cirstea, B.-I. *Contributions to handwriting recognition using deep neural networks and quantum computing*. Theses, Télécom ParisTech (2018). URL https://hal.archives-ouvertes.fr/tel-02412658.

[278] Elkenawy, A., El-Nagar, A. M., El-Bardini, M. & El-Rabaie, N. M. Full-state neural network observer-based hybrid quantum diagonal recurrent neural network adaptive tracking control. *Neural Computing and Applications* (2021). URL https://doi.org/10.1007/s00521-020-05685-x.

[279] Zak, M. & Williams, C. P. Quantum recurrent networks for simulating stochastic processes. In Williams, C. P. (ed.) *Quantum Computing and Quantum Communications*, 75–88 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1999). URL https://link.springer.com/content/pdf/10.1007/3-540-49208-9.pdf.

[280] Wootters, W. K. & Zurek, W. H. A single quantum cannot be cloned. *Nature* **299**, 802–803 (1982). URL https://doi.org/10.1038/299802a0.

[281] Dieks, D. Communication by epr devices. *Physics Letters A* **92**, 271–272 (1982). URL https://doi.org/10.1016/0375-9601(82)90084-6.

[282] Dawes, R. L. Quantum neurodynamics: neural stochastic filtering with the schroedinger equation. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, vol. 1, 133–140 vol.1 (1992). URL https://ieeexplore.ieee.org/document/287237.

[283] Behera, L. & Sundaram, B. Stochastic filtering and speech enhancement using a recurrent quantum neural network. In *International Conference on Intelligent Sensing and Information Processing, 2004. Proceedings of*, 165–170 (2004). URL https://ieeexplore.ieee.org/document/1287645.

[284] Behera, L., Kar, I. & Elitzur, A. C. *Recurrent Quantum Neural Network and its Applications*, 327–350 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). URL https://doi.org/10.1007/3-540-36723-3_9.

[285] Gandhi, V., Prasad, G., Coyle, D., Behera, L. & McGinnity, T. M. Quantum neural network-based eeg filtering for a brain–computer interface. *IEEE Transactions on Neural Networks and Learning Systems* **25**, 278–288 (2014). URL https://ieeexplore.ieee.org/document/6575167.

[286] Gandhi, V. S. & McGinnity, T. Quantum neural network based surface emg signal filtering for control of robotic hand. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, 1–7 (2013). URL https://ieeexplore.ieee.org/document/6706781.

[287] Lu, W. *et al.* Recurrent neural network approach to quantum signal: coherent state restoration for continuous-variable quantum key distribution. *Quantum Information Processing* **17**, 109 (2018). URL https://doi.org/10.1007/s11128-018-1877-y.

[288] Chen, S. Y.-C., Yoo, S. & Fang, Y.-L. L. Quantum long short-term memory (2020). URL https://arxiv.org/abs/2009.01783. 2009.01783.

[289] Chen, Y., Li, F., Wang, J., Tang, B. & Zhou, X. Quantum recurrent encoder–decoder neural network for performance trend prediction of rotating machinery. *Knowledge-Based Systems* **197**, 105863 (2020). URL https://www.sciencedirect.com/science/article/pii/S0950705120302252.

[290] Takaki, Y., Mitarai, K., Negoro, M., Fujii, K. & Kitagawa, M. Learning temporal data with a variational quantum recurrent neural network. *Phys. Rev. A* **103**, 052414 (2021). URL https://link.aps.org/doi/10.1103/PhysRevA.103.052414.

[291] Bausch, J. Recurrent quantum neural networks. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F. & Lin, H. (eds.) *Advances in Neural Information Processing Systems*, vol. 33, 1368–1379 (Curran Associates, Inc., 2020). URL https://proceedings.neurips.cc/paper/2020/file/0ec96be397dd6d3cf2fecb4a2d627c1c-Paper.pdf.

[292] Choi, J., Oh, S. & Kim, J. A tutorial on quantum graph recurrent neural network (qgrnn). In *2021 International Conference on Information Networking (ICOIN)*, 46–49 (2021). URL https://ieeexplore.ieee.org/document/9333917.

[293] Fujii, K. & Nakajima, K. *Reservoir Computing*, chap. Quantum Reservoir Computing: A Reservoir Approach Toward Quantum Machine Learning on Near-Term Quantum Devices. Natural Computing Series (Springer, Singapore, 2020). URL https://doi.org/10.1007/978-981-13-1687-6_18.

[294] Nokkala, J. *et al.* Gaussian states of continuous-variable quantum systems provide universal and versatile reservoir computing. *Communications Physics* **4**, 53 (2021). URL https://doi.org/10.1038/s42005-021-00556-w.

[295] Nakajima, K., Fujii, K., Negoro, M., Mitarai, K. & Kitagawa, M. Boosting computational power through spatial multiplexing in quantum reservoir computing. *Phys. Rev. Applied* **11**, 034021 (2019). URL https://link.aps.org/doi/10.1103/PhysRevApplied.11.034021.

[296] Mujal, P. *et al.* Opportunities in quantum reservoir computing and extreme learning machines. *Advanced Quantum Technologies* **4** (2021). URL https://doi.org/10.1002/qute.202100027.

[297] Wilde, M. M. Cambridge University Press (2019). URL http://markwilde.com/qit-notes.pdf.

[298] Kumar, S. Wishart and random density matrices: Analytical results for the mean-square hilbert-schmidt distance. *Physical Review A* **102**, 012405 (2020). URL http://dx.doi.org/10.1103/PhysRevA.102.012405.

[299] Addanki, S., Amiri, I. & Yupapin, P. Review of optical fibers-introduction and applications in fiber lasers. *Results in Physics* **10**, 743–750 (2018). URL https://www.sciencedirect.com/science/article/pii/S2211379718314268.

[300] Lu, P. *et al.* Distributed optical fiber sensing: Review and perspective. *Applied Physics Reviews* **6** (2019). URL https://pubs.aip.org/aip/apr/article/6/4/041302/124295.

[301] Senior, J. M. *Optical fiber communications* (Prentice Hall, 2009), 3 edn. Previous ed.: 1992.

[302] Ke, H. *et al.* Self-learning control for wavefront sensorless adaptive optics system through deep reinforcement learning. *Optik* **178**, 785–793 (2019). URL https://www.sciencedirect.com/science/article/pii/S0030402618314785.

[303] Nousiainen, J., Rajani, C., Kasper, M. & Helin, T. Adaptive optics control using model-based reinforcement learning. *Optics Express* **29**, 15327–15344 (2021). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-29-10-15327&id=450708.

[304] Durech, E., Newberry, W., Franke, J. & Sarunic, M. V. Wavefront sensor-less adaptive optics using deep reinforcement learning. *Biomedical optics express* **12**, 5423–5438 (2021). URL https://opg.optica.org/boe/fulltext.cfm?uri=boe-12-9-5423&id=455998.

[305] Landman, R., Haffert, S. Y., Radhakrishnan, V. M. & Keller, C. U. Self-optimizing adaptive optics control with reinforcement learning for high-contrast imaging. *Journal of Astronomical Telescopes, Instruments, and Systems* **7**, 039002–039002 (2021). URL https://doi.org/10.1117/1.JATIS.7.3.039002.

[306] Nousiainen, J. *et al.* Toward on-sky adaptive optics control using reinforcement learning-model-based policy optimization for adaptive optics. *Astronomy & Astrophysics* **664**, A71 (2022). URL https://www.aanda.org/articles/aa/abs/2022/08/aa43311-22/aa43311-22.html.

[307] Nousiainen, J. *et al.* Advances in model-based reinforcement learning for adaptive optics control. In *Adaptive Optics Systems VIII*, vol. 12185, 882–891 (SPIE, 2022). URL https://www.spiedigitallibrary.org/conference-proceedings-of-spie/12185/1218520/Advances-in-model-based-reinforcement-learning-for-adaptive-optics-control/10.1117/12.2630317.short#_=_.

[308] Kiran, Y., Venkatesh, T. & Murthy, C. S. R. A reinforcement learning framework for path selection and wavelength selection in optical burst switched networks. *IEEE Journal on Selected Areas in Communications* **25**, 18–26 (2007). URL https://ieeexplore.ieee.org/abstract/document/4395244.

[309] Suárez-Varela, J. *et al.* Routing in optical transport networks with deep reinforcement learning. *Journal of Optical Communications and Networking* **11**, 547–558 (2019). URL https://ieeexplore.ieee.org/abstract/document/8847548.

[310] Chen, X., Proietti, R. & Yoo, S. B. Building autonomic elastic optical networks with deep reinforcement learning. *IEEE Communications Magazine* **57**, 20–26 (2019). URL https://ieeexplore.ieee.org/abstract/document/8875708.

[311] Chen, X. *et al.* DeepRMSA: A deep reinforcement learning framework for routing, modulation and spectrum assignment in elastic optical networks. *Journal of Lightwave Technology* **37**, 4155–4163 (2019). URL https://opg.optica.org/jlt/abstract.cfm?uri=jlt-37-16-4155.

[312] Luo, X. *et al.* Leveraging double-agent-based deep reinforcement learning to global optimization of elastic optical networks with enhanced survivability. *Optics express* **27**, 7896–7911 (2019). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-27-6-7896&id=406937.

[313] Troia, S., Alvizu, R. & Maier, G. Reinforcement learning for service function chain reconfiguration in NFV-SDN metro-core optical networks. *IEEE Access* **7**, 167944–167957 (2019). URL https://ieeexplore.ieee.org/abstract/document/8901169.

[314] Li, X., Hu, X., Zhang, R. & Yang, L. Routing protocol design for underwater optical wireless sensor networks: A multiagent reinforcement learning approach. *IEEE Internet of Things Journal* **7**, 9805–9818 (2020). URL https://ieeexplore.ieee.org/abstract/document/9076600.

[315] Natalino, C. & Monti, P. The optical RL-gym: An open-source toolkit for applying reinforcement learning in optical networks. In *22nd International Conference on Transparent Optical Networks (ICTON)*, 1–5 (IEEE, 2020). URL https://ieeexplore.ieee.org/abstract/document/9203239.

[316] Yu, X. *et al.* Real-time adaptive optical self-interference cancellation for in-band full-duplex transmission using SARSA (λ) reinforcement learning. *Optics Express* **31**, 13140–13153 (2023). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-31-8-13140&id=528833.

[317] Sajedian, I., Badloe, T. & Rho, J. Optimisation of colour generation from dielectric nanostructures using reinforcement learning. *Optics express* **27**, 5874–5883 (2019). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-27-4-5874&id=405163.

[318] Jiang, A., Osamu, Y. & Chen, L. Multilayer optical thin film design with deep Q-learning. *Scientific reports* **10**, 12780 (2020). URL https://www.nature.com/articles/s41598-020-69754-w.

[319] Wang, H., Zheng, Z., Ji, C. & Guo, L. J. Automated multi-layer optical design via deep reinforcement learning. *Machine Learning: Science and Technology* **2**, 025013 (2021). URL https://iopscience.iop.org/article/10.1088/2632-2153/abc327/meta.

[320] Wankerl, H., Stern, M. L., Mahdavi, A., Eichler, C. & Lang, E. W. Parameterized reinforcement learning for optical system optimization. *Journal of Physics D: Applied Physics* **54**, 305104 (2021). URL https://iopscience.iop.org/article/10.1088/1361-6463/abfddb/meta.

[321] Sun, C., Kaiser, E., Brunton, S. L. & Kutz, J. N. Deep reinforcement learning for optical systems: A case study of mode-locked lasers. *Machine Learning: Science and Technology* **1** (2020). URL https://iopscience.iop.org/article/10.1088/2632-2153/abb6d6/meta.

[322] Tünnermann, H. & Shirakawa, A. Deep reinforcement learning for tiled aperture beam combining in a simulated environment. *Journal of Physics: Photonics* **3**, 015004 (2021). URL https://iopscience.iop.org/article/10.1088/2515-7647/abcd83/meta.

[323] Abuduweili, A., Wang, J., Yang, B., Wang, A. & Zhang, Z. Reinforcement learning based robust control algorithms for coherent pulse stacking. *Optics Express* **29**, 26068–26081 (2021). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-29-16-26068&id=453824.

[324] Abuduweili, A. & Liu, C. An optical controlling environment and reinforcement learning benchmarks (2022). URL https://arxiv.org/abs/2203.12114. arXiv:2203.12114.

[325] Sorokin, D., Ulanov, A., Sazhina, E. & Lvovsky, A. Interferobot: aligning an optical interferometer by a reinforcement learning agent. In *Advances in Neural Information Processing Systems*, vol. 33 (2020). URL https://proceedings.neurips.cc/paper_files/paper/2020/file/99ba5c4097c6b8fef5ed774a1a6714b8-Paper.pdf.

[326] Mukund, N. *et al.* Neural sensing and control in a kilometer-scale gravitational-wave observatory. *Physical Review Applied* **20**, 064041 (2023). URL https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.20.064041.

[327] Praeger, M., Xie, Y., Grant-Jacob, J. A., Eason, R. W. & Mills, B. Playing optical tweezers with deep reinforcement learning: in virtual, physical and augmented environments. *Machine Learning: Science and Technology* **2**, 035024 (2021). URL https://iopscience.iop.org/article/10.1088/2632-2153/abf0f6/meta.

[328] Shpakovych, M. *et al.* Experimental phase control of a 100 laser beam array with quasi-reinforcement learning of a neural network in an error reduction loop. *Optics Express* **29**, 12307–12318 (2021). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-29-8-12307&id=449939.

[329] Kuprikov, E., Kokhanovskiy, A., Serebrennikov, K. & Turitsyn, S. Deep reinforcement learning for self-tuning laser source of dissipative solitons. *Scientific Reports* **12**, 7185 (2022). URL https://www.nature.com/articles/s41598-022-11274-w.

[330] Tünnermann, H. & Shirakawa, A. Deep reinforcement learning for coherent beam combining applications. *Optics express* **27**, 24223–24230 (2019). URL https://opg.optica.org/oe/fulltext.cfm?uri=oe-27-17-24223&id=416642.

[331] Valensise, C. M., Giuseppi, A., Cerullo, G. & Polli, D. Deep reinforcement learning control of white-light continuum generation. *Optica* **8**, 239–242 (2021). URL https://opg.optica.org/optica/fulltext.cfm?uri=optica-8-2-239&id=447607.

[332] Chiappa, A. S., Marin Vargas, A., Huang, A. & Mathis, A. Latent exploration for reinforcement learning. In Oh, A. *et al.* (eds.) *Advances in Neural Information Processing Systems*, vol. 36, 56508–56530 (Curran Associates, Inc., 2023). URL https://proceedings.neurips.cc/paper_files/paper/2023/file/b0ca717599b7ba84d5e4f4c8b1ef6657-Paper-Conference.pdf.

[333] Eberhard, O., Hollenstein, J., Pinneri, C. & Martius, G. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *Deep Reinforcement Learning Workshop NeurIPS 2022* (2022). URL https://openreview.net/forum?id=imxyoQIC5XT.

[334] Hollenstein, J., Auddy, S., Saveriano, M., Renaudo, E. & Piater, J. Action noise in off-policy deep reinforcement learning: Impact on exploration and performance. *Transactions on Machine Learning Research* (2022). URL https://openreview.net/forum?id=NljBlZ6hmG. Survey Certification.

[335] Zhang, Y. & Van Hoof, H. Deep coherent exploration for continuous control. In Meila, M. & Zhang, T. (eds.) *Proceedings of the 38th International Conference on Machine Learning*, vol. 139 of *Proceedings of Machine Learning Research*, 12567–12577 (PMLR, 2021). URL https://proceedings.mlr.press/v139/zhang21t.html.

[336] Plappert, M. *et al.* Parameter space noise for exploration. In *International Conference on Learning Representations* (2018). URL https://openreview.net/forum?id=ByBAl2eAZ.

[337] Knigge, A., Rahmel, M. & Knothe, C. *Fiber Coupling to Polarization-Maintaining Fibers and Collimation*. Schäfter+Kirchhoff GmbH. URL https://www.sukhamburg.com/documents/Article_FibercouplingNAe2.pdf.

[338] Brockman, G. *et al.* Openai gym (2016). URL https://arxiv.org/abs/1606.01540. Licensed under The MIT License, Version: gymnasium 0.29.1, arXiv:1606.01540.

[339] Bruce, P. C. & Bruce, A. *Practical Statistics for Data Scientists* (O'Reilly Media, Inc., 2017), first edition. edn. URL https://www.oreilly.com/library/view/practical-statistics-for/9781491952955/.

[340] Raffin, A. *et al.* Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research* **22**, 1–8 (2021). URL http://jmlr.org/papers/v22/20-1364.html. Licensed under The MIT License, Version 2.3.0.

[341] Team, T. P. D. pandas-dev/pandas: Pandas (2020). URL https://doi.org/10.5281/zenodo.3509134. Licensed under BSD 3-Clause License, Version 1.4.4.

[342] McKinney, W. Data structures for statistical computing in python. In van der Walt, S. & Millman, J. (eds.) *Proceedings of the 9th Python in Science Conference*, 56 – 61 (2010). URL http://conference.scipy.org.s3.amazonaws.com/proceedings/scipy2010/pdfs/mckinney.pdf.

[343] Narvekar, S. *et al.* Curriculum learning for reinforcement learning domains: A framework and survey (2020). URL https://arxiv.org/abs/2003.04960. 2003.04960.

[344] Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362 (2020). URL https://doi.org/10.1038/s41586-020-2649-2. Licensed under modified BSD license, Version 1.26.2.

[345] Abadi, M. *et al.* TensorFlow: Large-scale machine learning on heterogeneous systems (2015). URL https://www.tensorflow.org/. Licensed under Apache License, Version 2.15.1.

[346] Waskom, M. L. Seaborn: Statistical data visualization. *Journal of Open Source Software* **6**, 3021 (2021). URL https://doi.org/10.21105/joss.03021. Licensed under BSD 3-Clause "New" or "Revised" License, Version 0.11.2.

[347] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering* **9**, 90–95 (2007). URL https://ieeexplore.ieee.org/document/4160265. Licensed under a BSD style license, Version 3.5.2.

[348] Shkarin, A. pylablib (2024). URL https://pylablib.readthedocs.io/en/latest/. Licensed under Creative Commons Attribution 4.0 International, Version 1.4.2.

[349] Thorlabs kinesis. URL https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=10285. Licensed by Thorlabs, All Rights Reserved, Version 1.14.44.

[350] Grecco, H. E., Dartiailh, M. C., Thalhammer-Thurner, G., Bronger, T. & Bauer, F. Pyvisa: the python instrumentation package. *Journal of Open Source Software* **8**, 5304 (2023). URL https://doi.org/10.21105/joss.05304. Licensed under The MIT License, Version 1.14.1.

[351] Keysight connection expert. URL https://www.keysight.com/us/en/lib/software-detail/computer-software/io-libraries-suite-downloads-2175637.html. Licensed as Commercial computer software, Revision 2023 Update 1.

[352] JianAo, Z. safe-exit. URL https://pypi.org/project/safe-exit/#:~:text=SafeExitisaPython,calledwhentheprogramexits. Licensed under MIT License, Version 0.1.2.

[353] Char, I. & Schneider, J. PID-inspired inductive biases for deep reinforcement learning in partially observable control tasks. In *Advances in Neural Information Processing Systems*, vol. 36 (2023). URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/ba1c5356d9164bb64c446a4b690226b0-Abstract-Conference.html.

[354] Black, E. D. An introduction to pound–drever–hall laser frequency stabilization. *American Journal of Physics* **69**, 79–87 (2001). URL https://pubs.aip.org/aapt/ajp/article-abstract/69/1/79/1055569/An-introduction-to-Pound-Drever-Hall-laser?redirectedFrom=fulltext.

[355] Drever, R. W. *et al.* Laser phase and frequency stabilization using an optical resonator. *Applied Physics B* **31**, 97–105 (1983). URL https://link.springer.com/article/10.1007/bf00702605.

[356] Hansch, T. & Couillaud, B. Laser frequency stabilization by polarization spectroscopy of a reflecting reference cavity. *Optics communications* **35**, 441–444 (1980). URL https://www.sciencedirect.com/science/article/pii/0030401880900693.

[357] Heurs, M., Petersen, I. R., James, M. R. & Huntington, E. H. Homodyne locking of a squeezer. *Opt. Lett.* **34**, 2465–2467 (2009). URL https://opg.optica.org/ol/abstract.cfm?URI=ol-34-16-2465.

[358] Shaddock, D. A., Gray, M. B. & McClelland, D. E. Frequency locking a laser to an optical cavity by use of spatial mode interference. *Opt. Lett.* **24**, 1499–1501 (1999). URL https://opg.optica.org/ol/abstract.cfm?URI=ol-24-21-1499.

[359] Chabbra, N. *et al.* High stability laser locking to an optical cavity using tilt locking. *Opt. Lett.* **46**, 3199–3202 (2021). URL https://opg.optica.org/ol/abstract.cfm?URI=ol-46-13-3199.

[360] Vahlbruch, H. *et al.* Coherent control of vacuum squeezing in the gravitational-wave detection band. *Phys. Rev. Lett.* **97**, 011101 (2006). URL https://link.aps.org/doi/10.1103/PhysRevLett.97.011101.

[361] McKenzie, K. *et al.* Quantum noise locking. *Journal of Optics B: Quantum and Semiclassical Optics* **7**, S421 (2005). URL https://dx.doi.org/10.1088/1464-4266/7/10/032.

[362] Barry, J., Barry, D. T. & Aaronson, S. Quantum partially observable markov decision processes. *Physical Review A* **90**, 032311 (2014). URL https://doi.org/10.1103/PhysRevA.90.032311.

[363] Gordon, R. J. & Rice, S. A. Active control of the dynamics of atoms and molecules. *Annual Review of Physical Chemistry* **48**, 601–641 (1997). URL https://doi.org/10.1146/annurev.physchem.48.1.601.

[364] Shapiro, M. & Brumer, P. *Coherent Control of Atomic, Molecular, and Electronic Processes*, 287–345 (Elsevier, 2000). URL https://doi.org/10.1016/S1049-250X(08)60189-5.

[365] Tsang, M. & Caves, C. M. Coherent quantum-noise cancellation for optomechanical sensors. *Physical Review Letters* **105** (2010). URL http://dx.doi.org/10.1103/PhysRevLett.105.123601.

Curriculum Vitae

Name	Viktoria-Sophie Schmiesing
Date of birth	29.09.1997
Place of birth	Hannover, Germany

Academic Career

2020-2024	**PhD student** at Leibniz Universität Hannover Institute for Theoretical Physics Quantum Information Group, Prof. T. J. Osborne Research, IT administration
2018-2020	**Master of Science in Physics** at Leibniz Universität Hannover Thesis: "Quantum Learning Theory" supervised by Prof. T. J. Osborne
2015-2018	**Bachelor of Science in Physics** at Leibniz Universität Hannover Thesis: "Diffusion in 1D chains" supervised by Prof. H. Frahm
2015	**Abitur** at St. Ursula Schule Hannover

List of Publications

- D. Bondarenko, R. Salzmann, and V.-S. Schmiesing.
  Learning Quantum Processes with Memory – Quantum Recurrent Neural Networks.
  https://arxiv.org/abs/2301.08167 (2023)
- L. Richtmann, V.-S. Schmiesing, D. Wilken, J. Heine, A. Tranter, A. Anand, T. J. Osborne, and M. Heurs.
  Model-free Reinforcement Learning with Noisy Actions for Automated Experimental Control in Optics
  https://arxiv.org/abs/2405.15421 (2024)